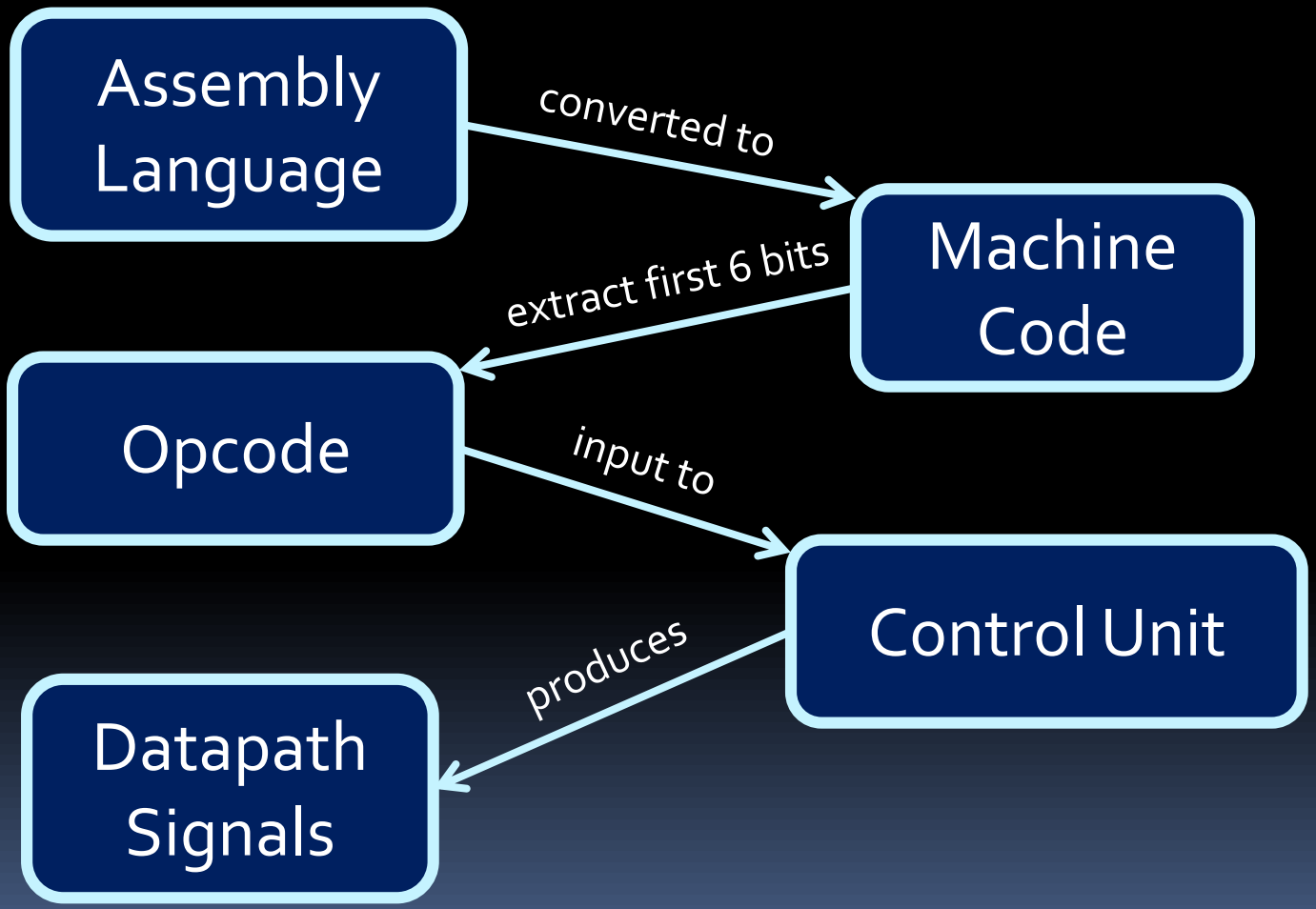
Assembly Language

* Created with contributions by Myrto Papadopoulou and Frank Plavec.

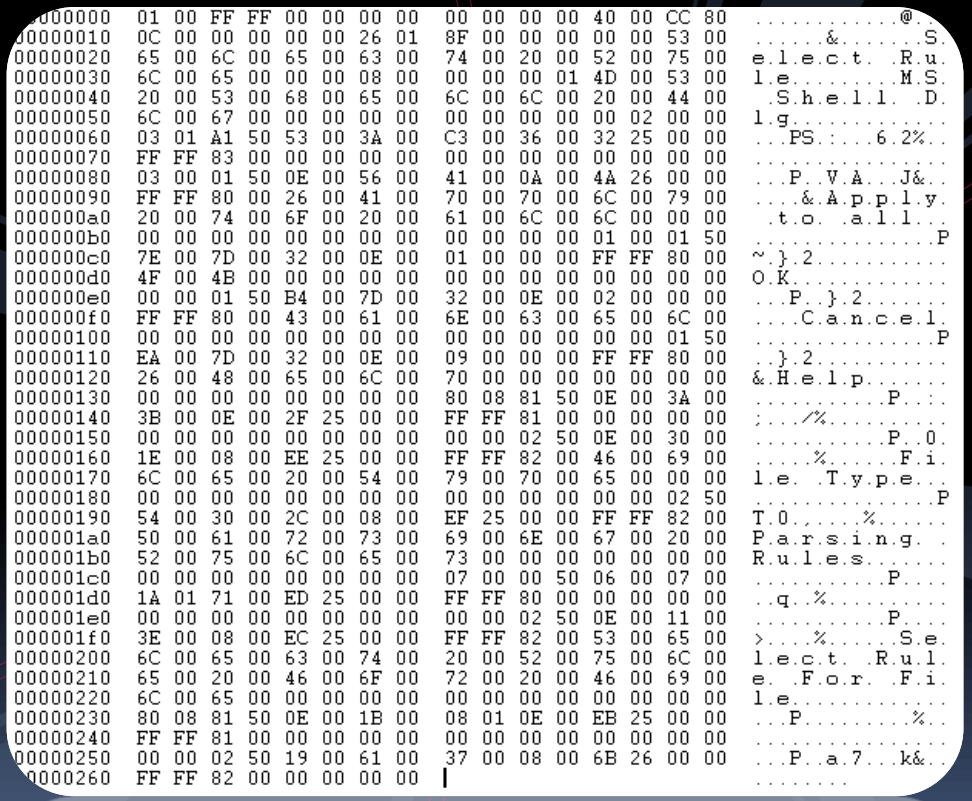
Programming the processor

- 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



```
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, 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
```

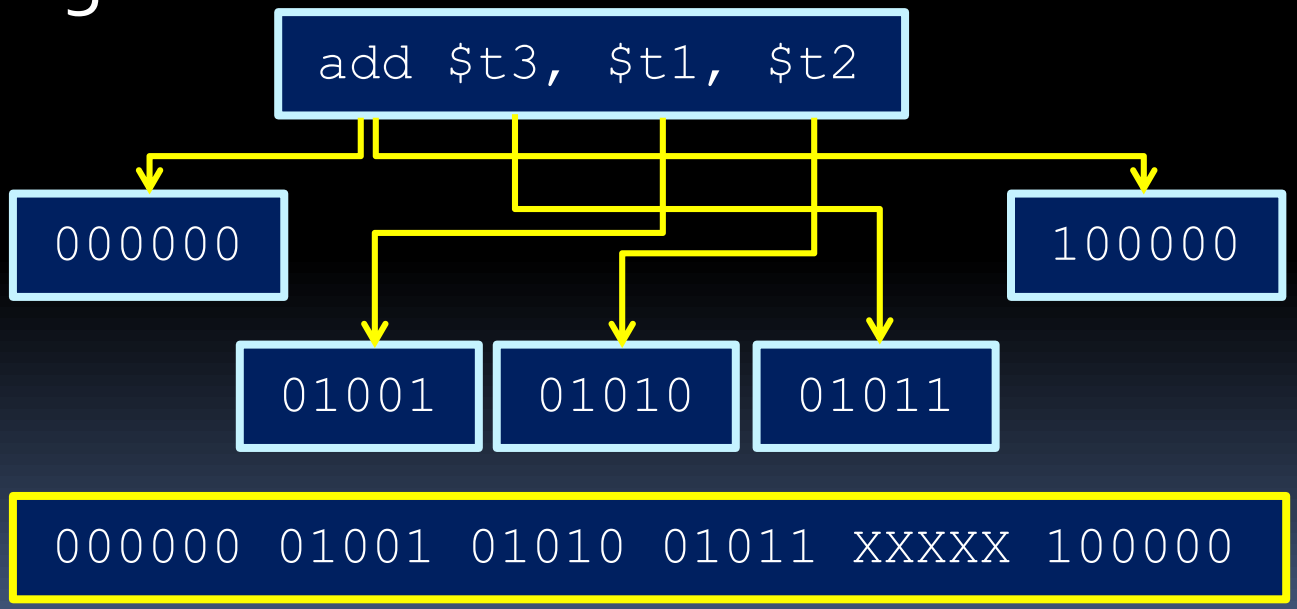
Note: There is a 1-to-1 mapping for all assembly code and machine code instructions!

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

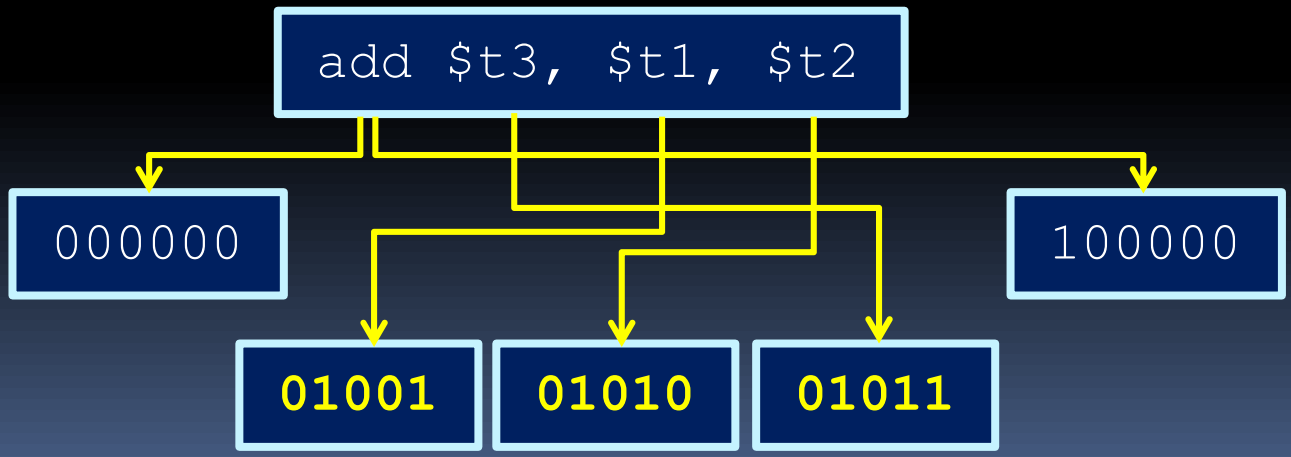
- The `add` instruction is one of many operations that processes two register values and stores the result in a third.
  - This is called an R-type instruction.
    - Any operations whose inputs and outputs are all registers are called R-type, even if they involve less than three registers.
    - e.g.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

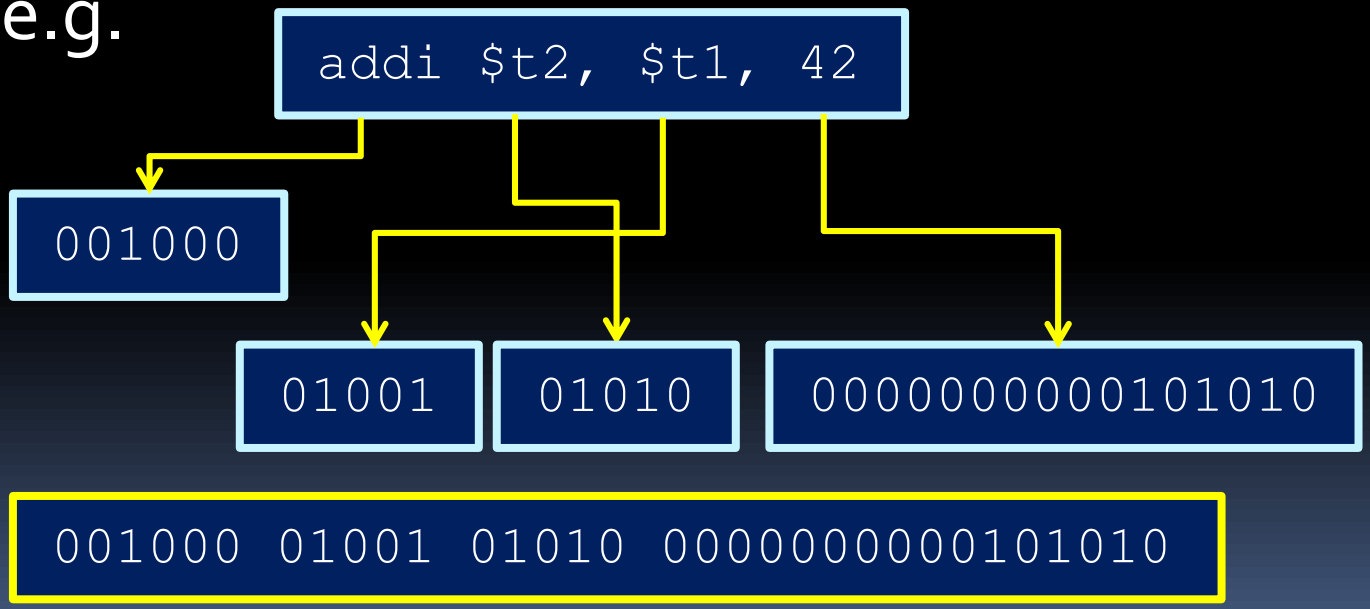
Filling in the blanks

- 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`).



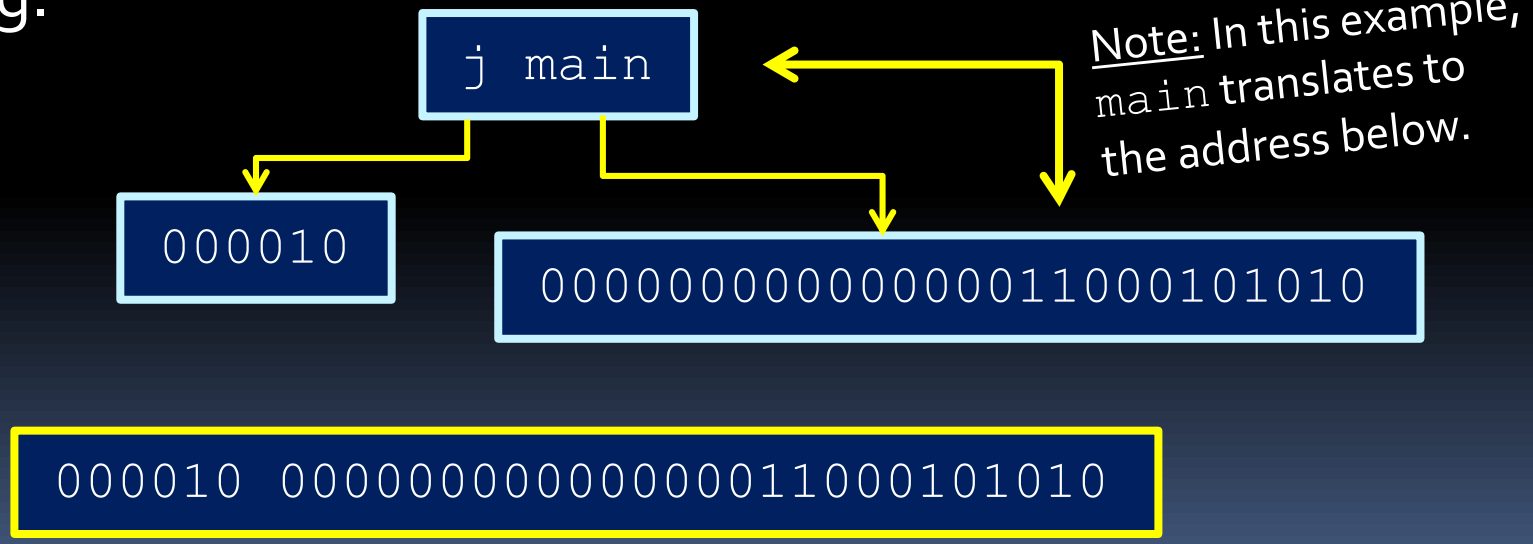
I-type instructions

- 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.



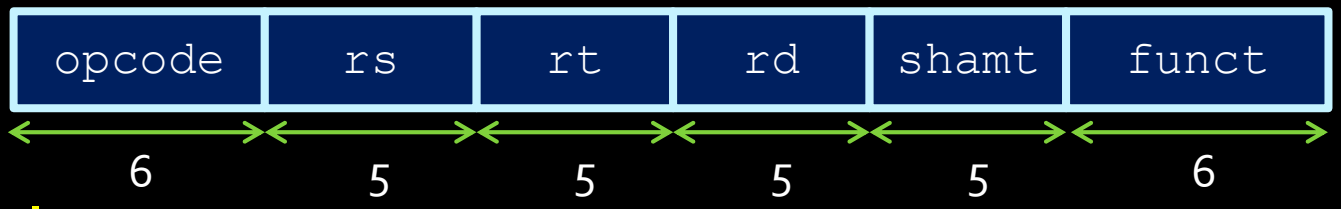
J-type instructions

- 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.



Review: MIPS instruction types

- R-type:



- I-type:

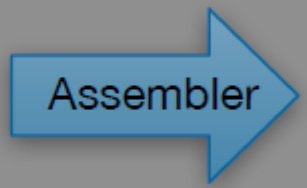
- J-type:

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

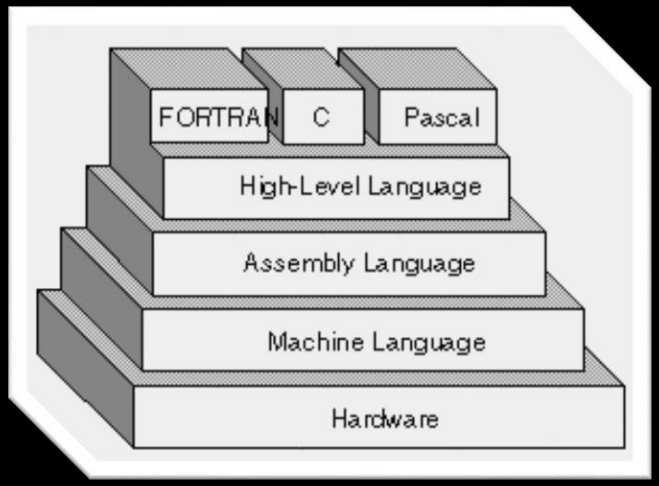
```
loop: lw   $t3, 0($t0)
      lw   $t4, 4($t0)
      add  $t2, $t3, $t4
      sw   $t2, 8($t0)
      addi $t0, $t0, 4
      addi $t1, $t1, -1
      bgtz $t1, loop
```




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

Assembly language

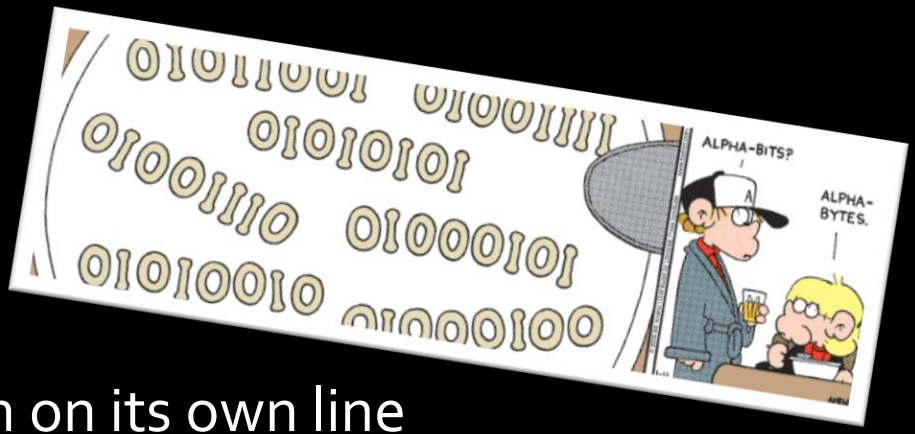
- Assembly language is the lowest-level language that you'll ever program in.
- Many compilers translate their high-level program commands into assembly commands, which are then converted into machine code and used by the processor.
- Note: There are multiple types of assembly language, especially for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



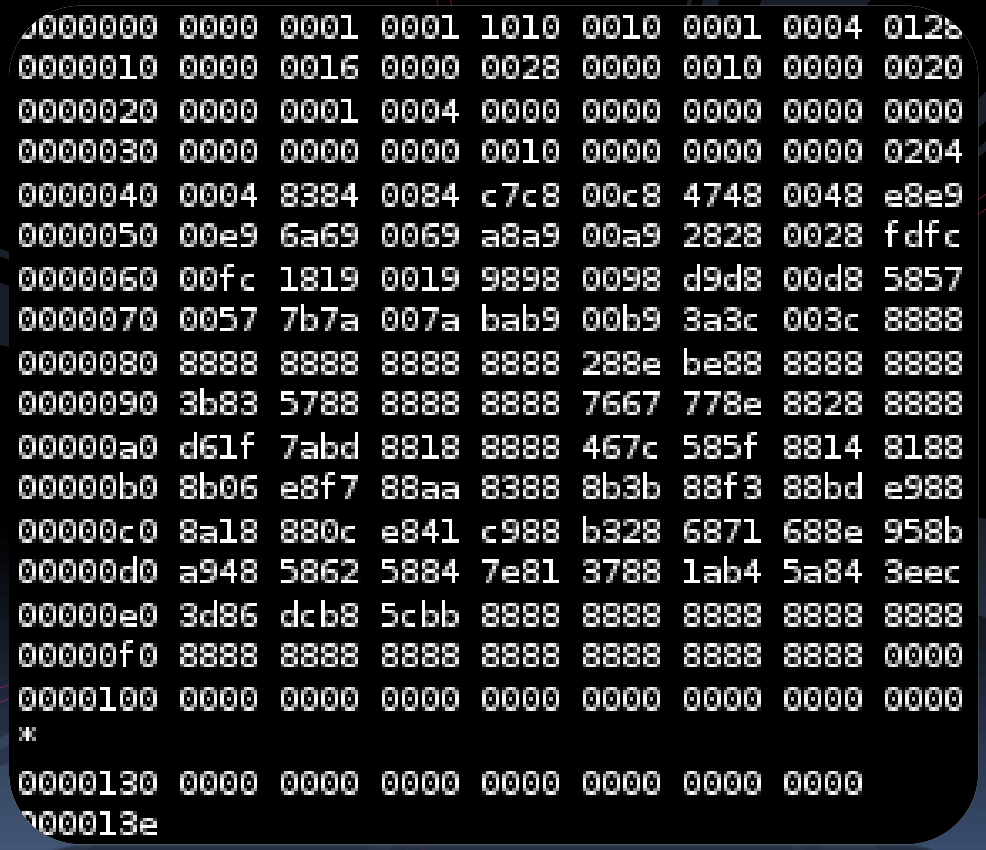
- Things to note about MIPS instructions:
  - Instruction are written `<instr> <parameters>`
  - Each instruction is written on its own line
  - All instructions are 32 bits (4 bytes) long
  - Instruction addresses are measured in bytes, starting from the instruction at address `0`.
    - Therefore,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

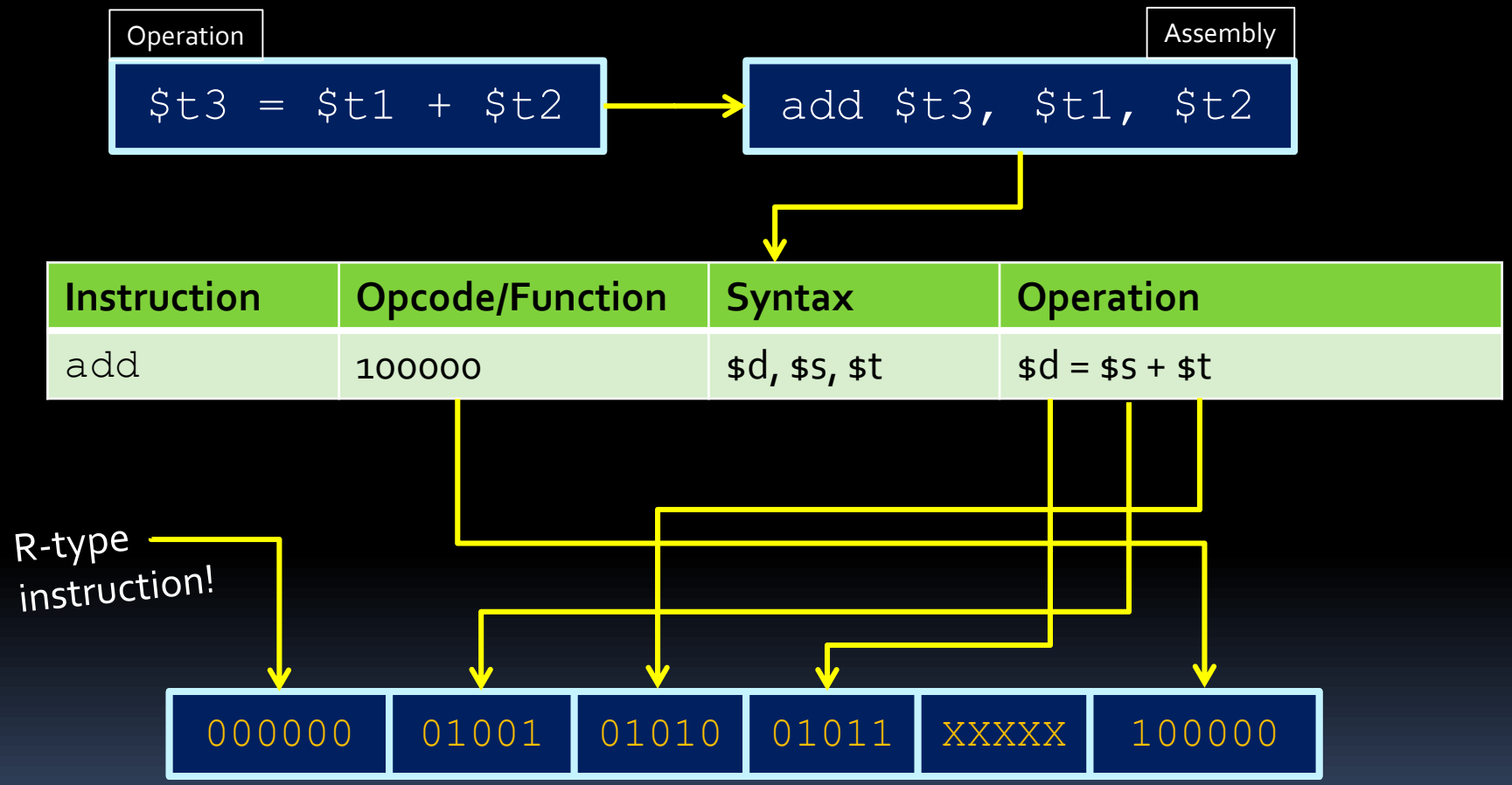


Arithmetic instructions

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

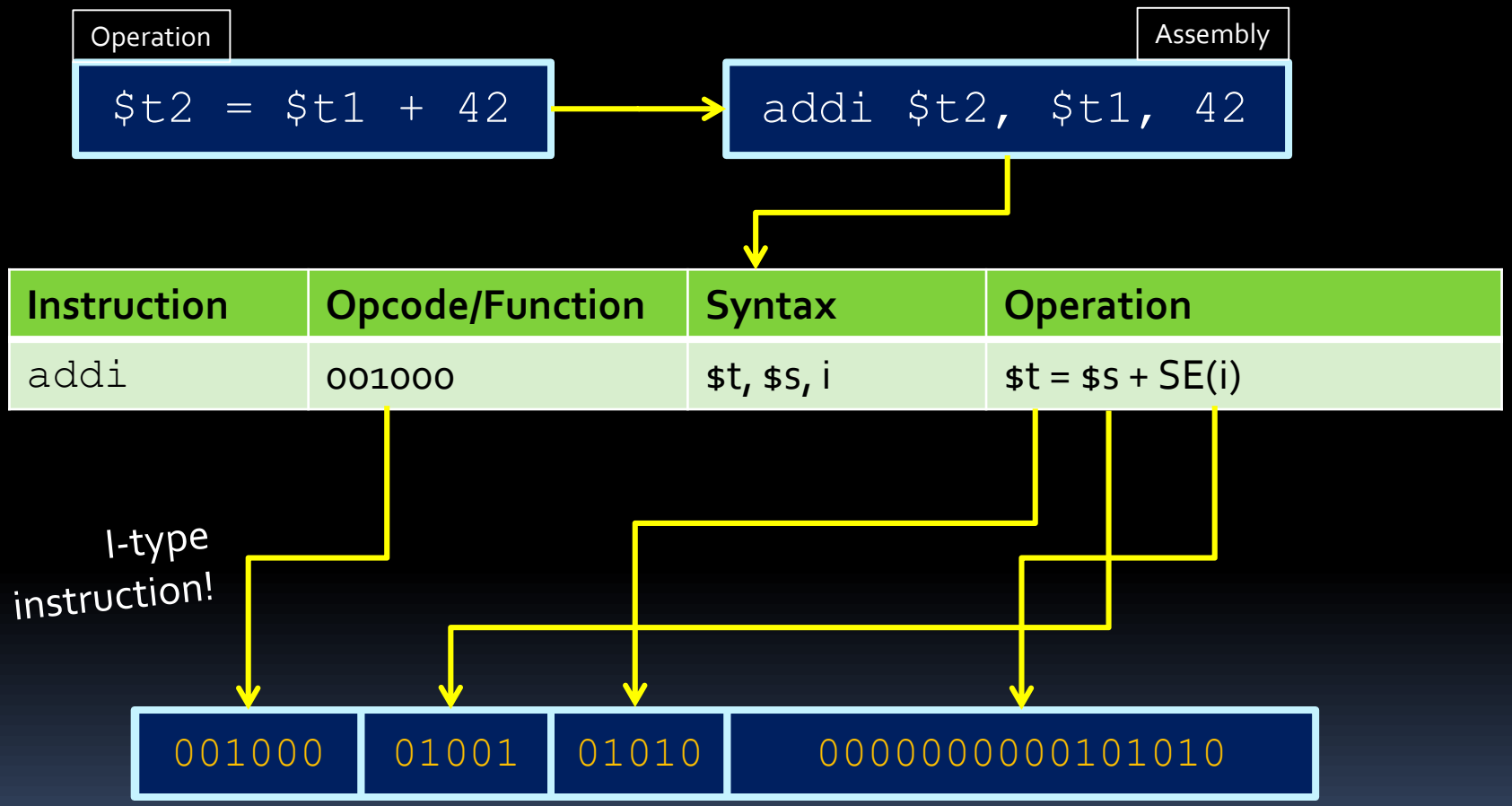
- `add, addu`
- `div, divu`
- `mult, multu`
- `sub, subu`

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 << 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

- Fibonacci sequence i

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

```
# 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
```

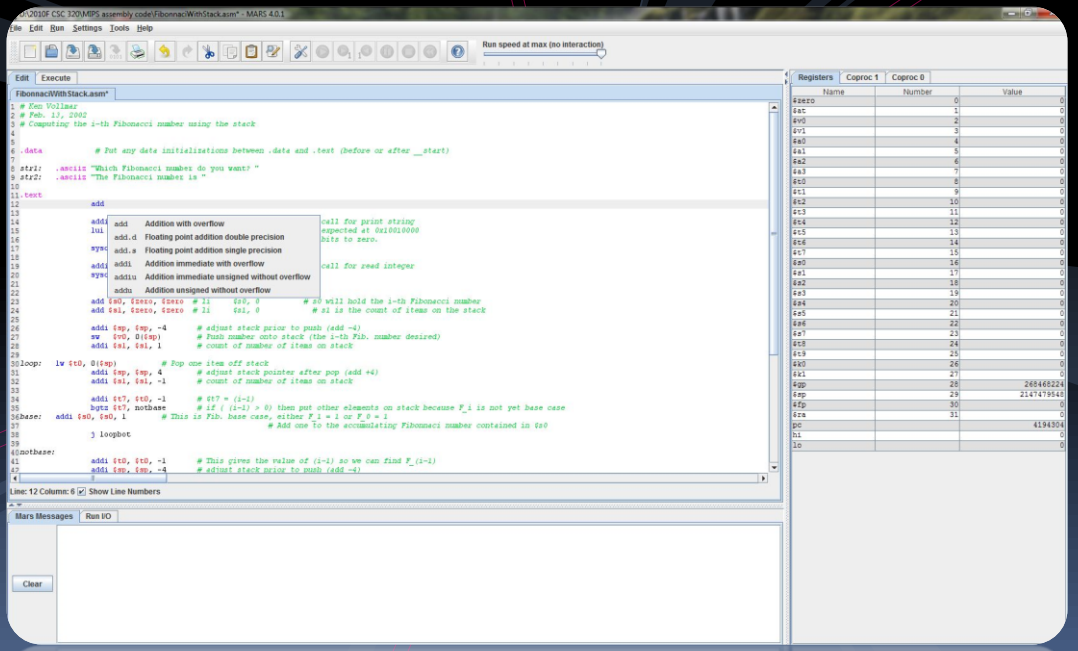
Making an assembly program

- 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 ☺

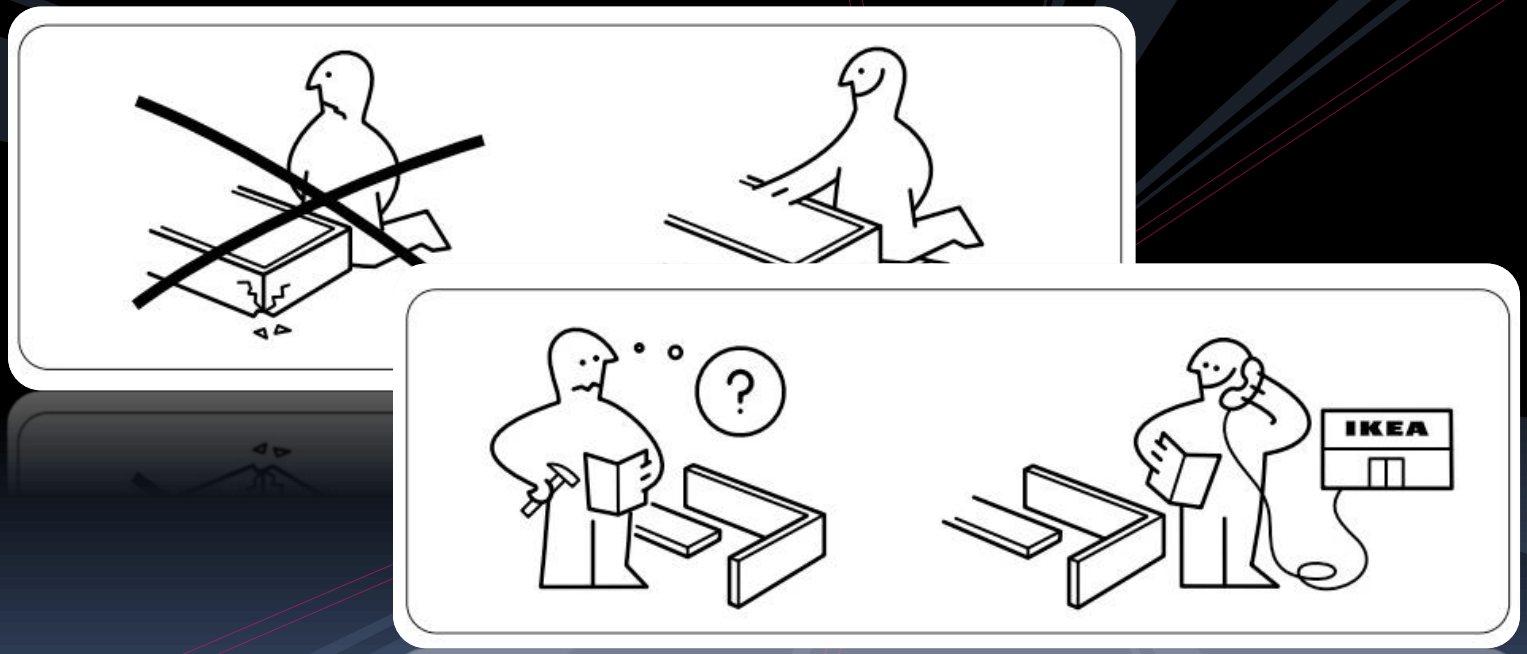
Simulating MIPS

aka: MARS

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



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	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.

`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.

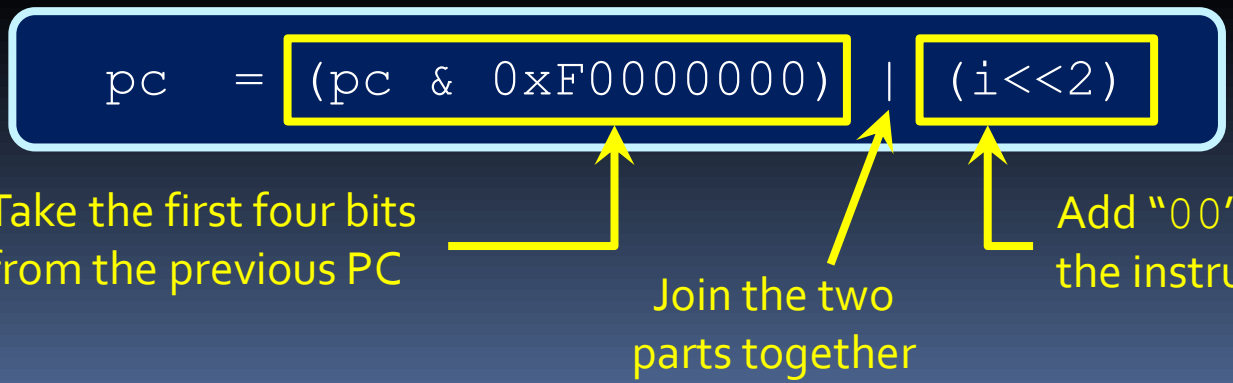
- If the first 6 bits are occupied by the opcode, the remaining bits aren't enough for a full 32-bit address!
- How do we get around this?

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:



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
        ...                    # 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

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

  - If relative to incremented PC:

```
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 (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 `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;
```

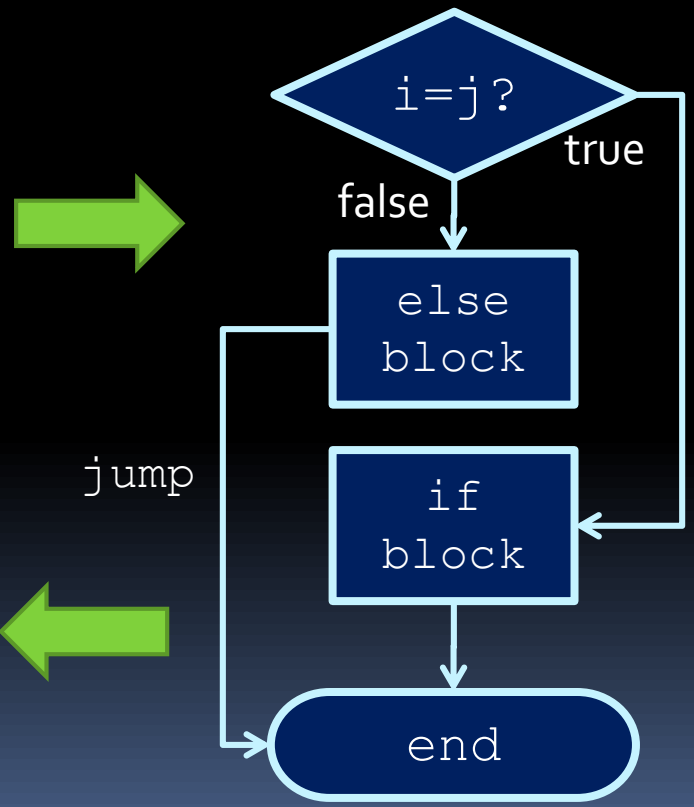
- Or branch on the else condition first:

```
#  $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;
```

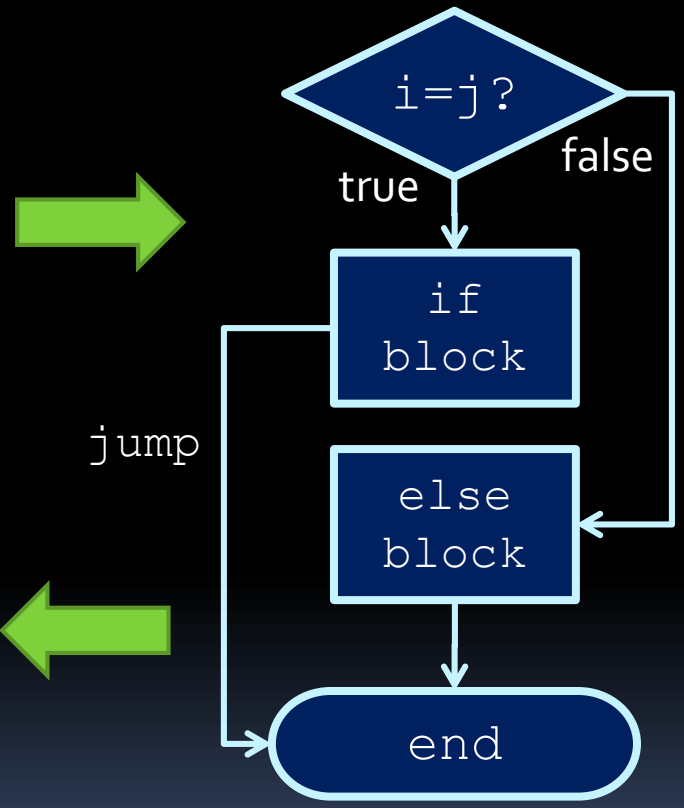



```
#   $t1 = i, $t2 = j
main:    beq  $t1, $t2, IF
         addi $t1, $t1, -1
         j END
IF:      addi $t1, $t1, 1
END:     add $t2, $t2, $t1
```

jump

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

Multiple `if` conditions

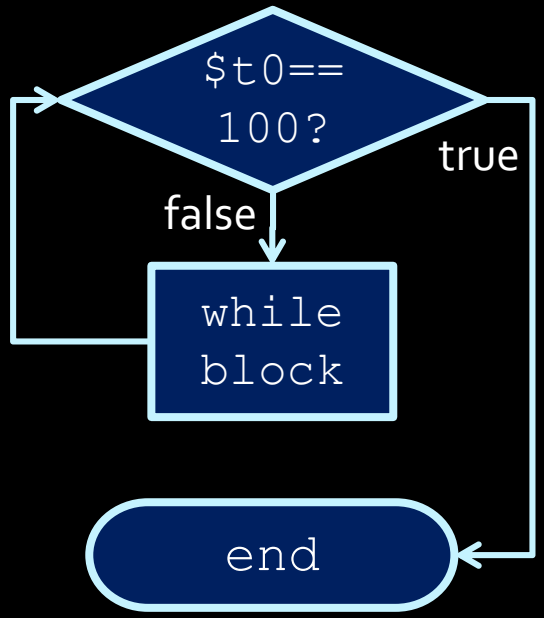
- How would this look if the condition changed?

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

- This translates to:

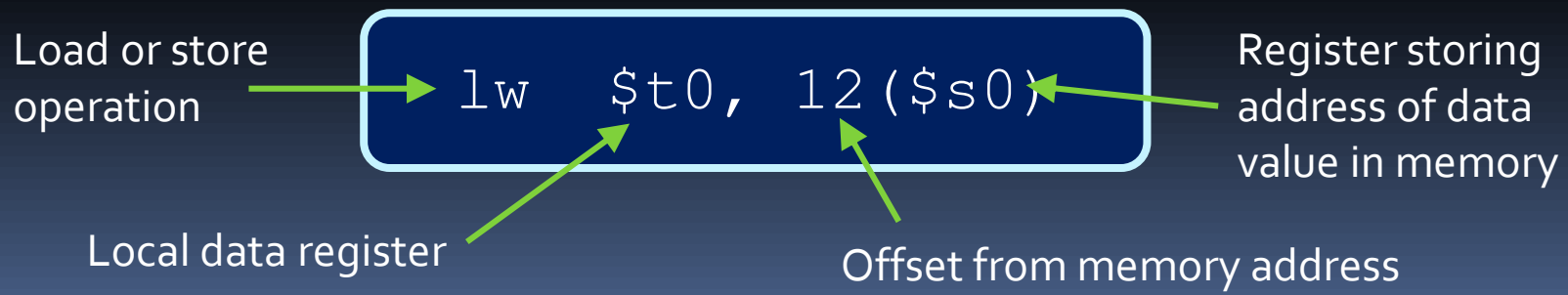
```
#  $t0 = i, $t1 = j
main:    add $t0, $zero, $zero        # set $t0 to 0
         add $t1, $zero, $zero        # set $t1 to 0
         addi $t9, $zero, 100         # set $t9 to 100
START:   beq $t0, $t9, EXIT           # branch if i==100
         add $t1, $t1, $t0            # j = j + i
UPDATE: 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

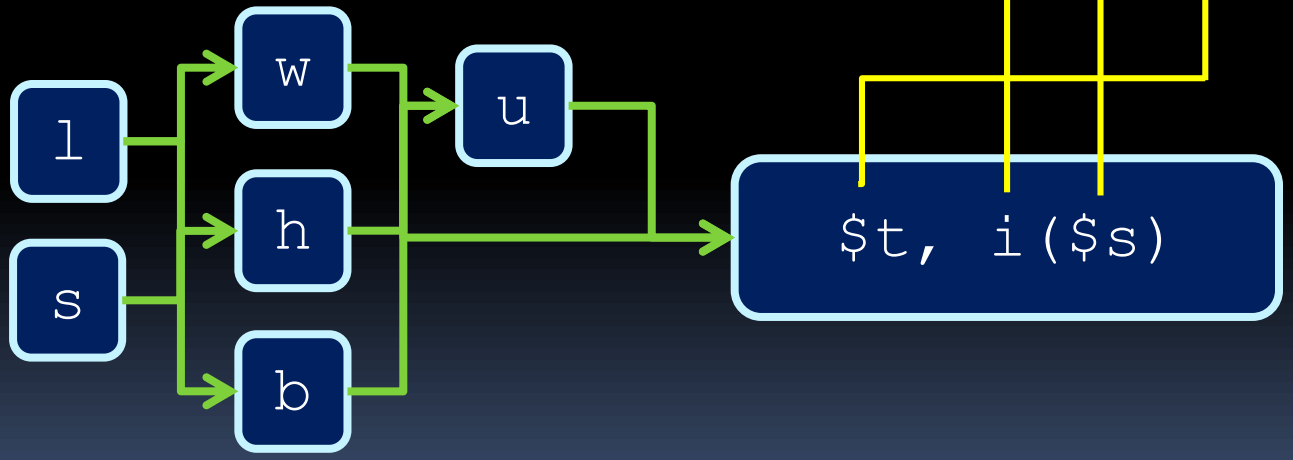
- The terms "load" and "store" are seen from the perspective of the processor, looking at memory.

- 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:

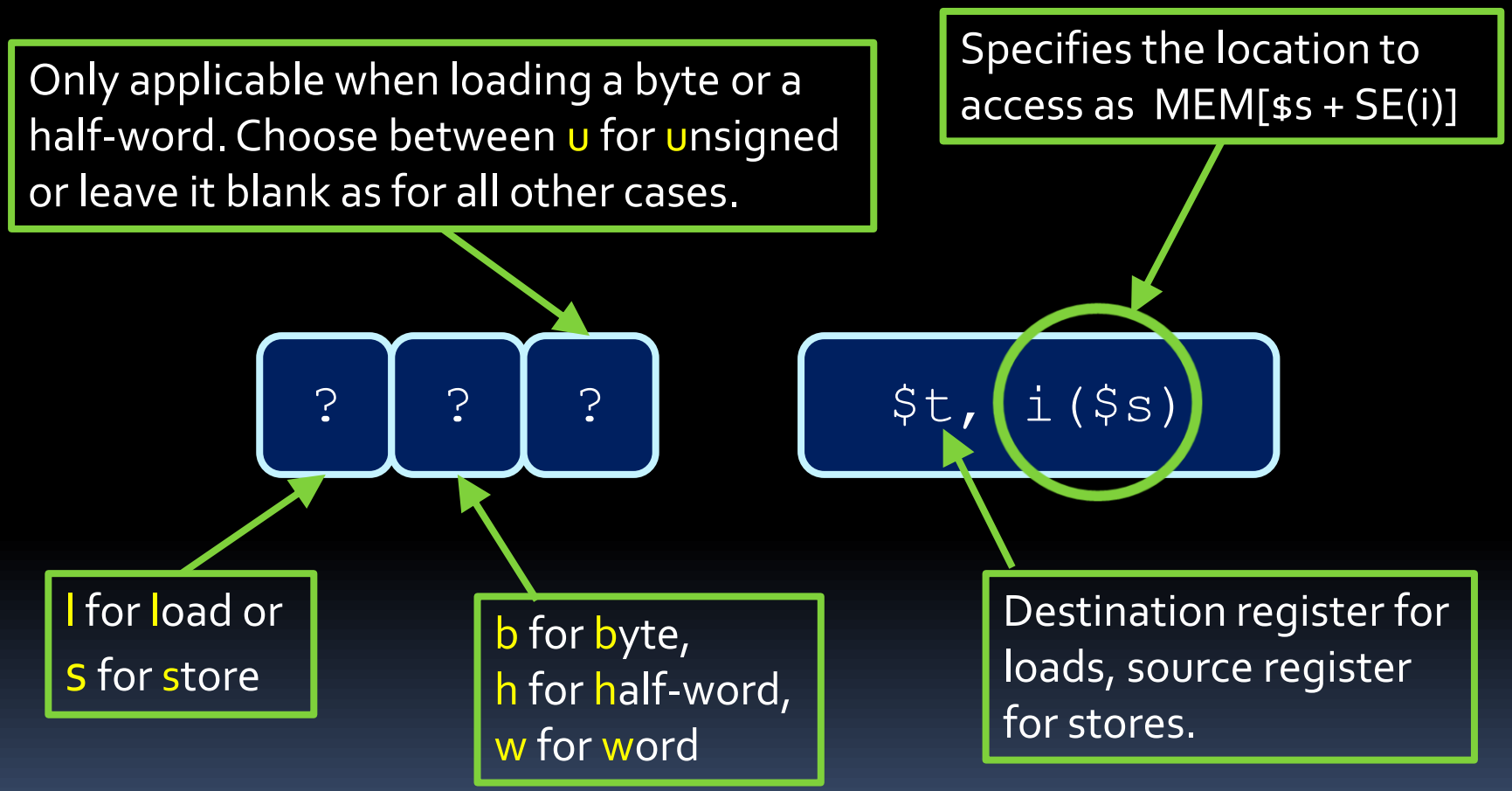


Load & store instructions

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".

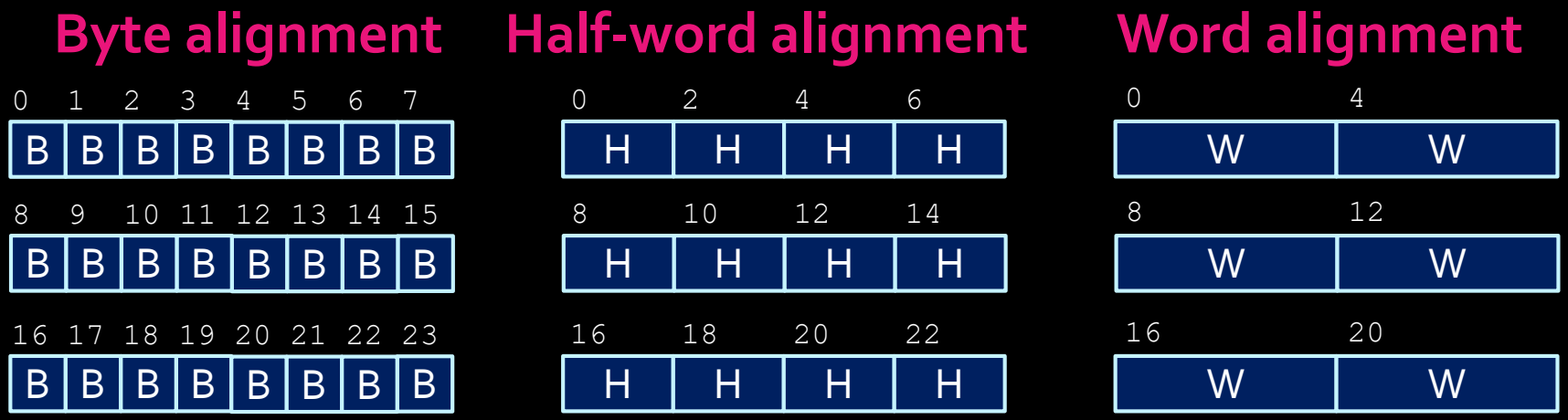
Memory Instructions in MIPS assembly



Alignment Requirements

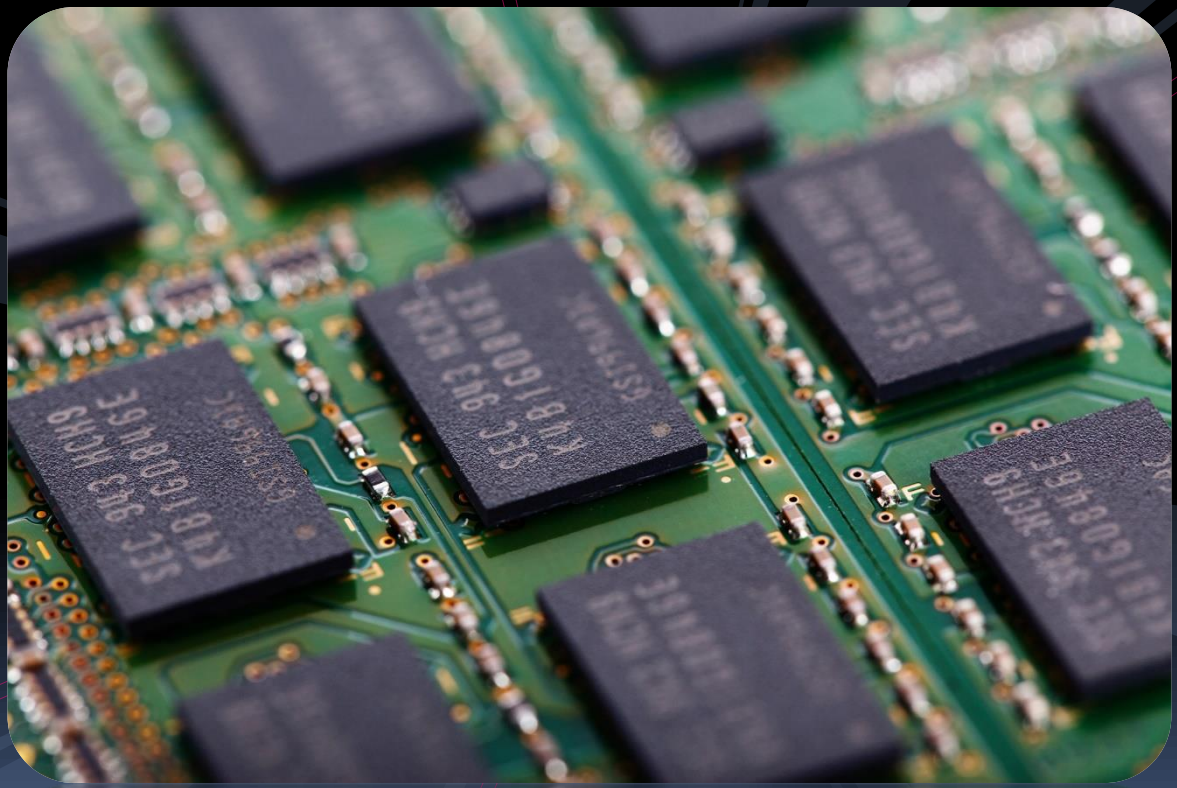
- Misaligned memory accesses result in errors.
  - Word accesses (i.e., addresses specified in a `lw` or `sw` instruction) should be word-aligned (divisible by 4).
  - Half-word accesses should only involve half-word aligned addresses (i.e., even addresses).
  - No constraints for byte accesses.

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

Big Endian:

Byte A	Byte B	Byte C	Byte D

Least significant byte

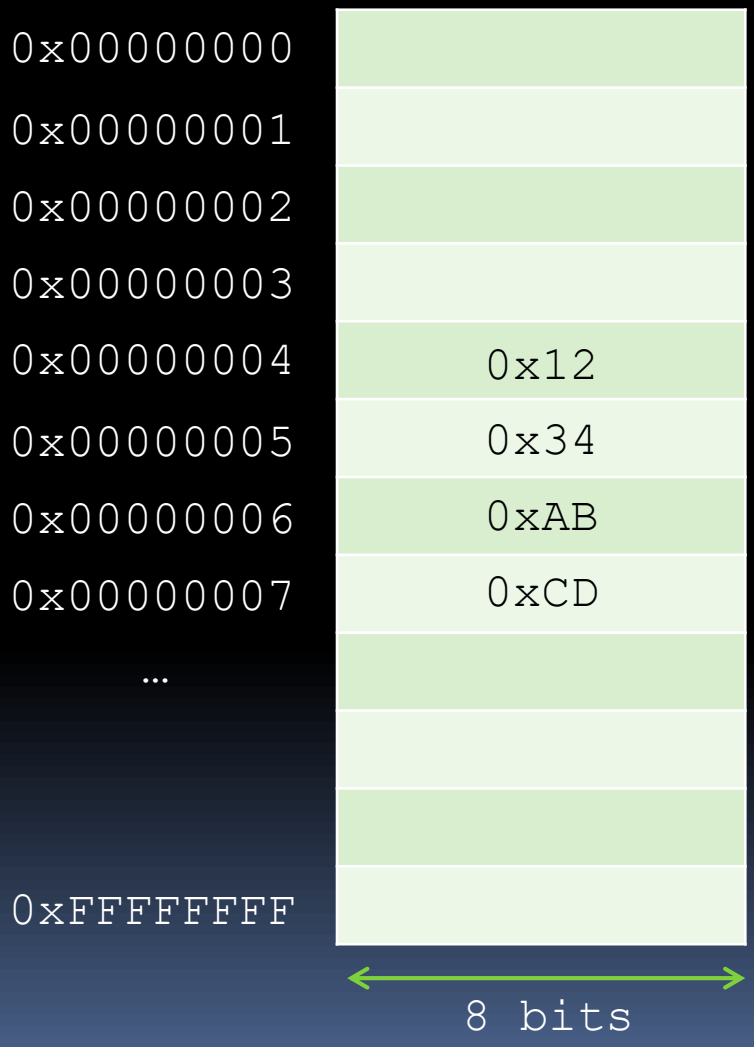
Little Endian:

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

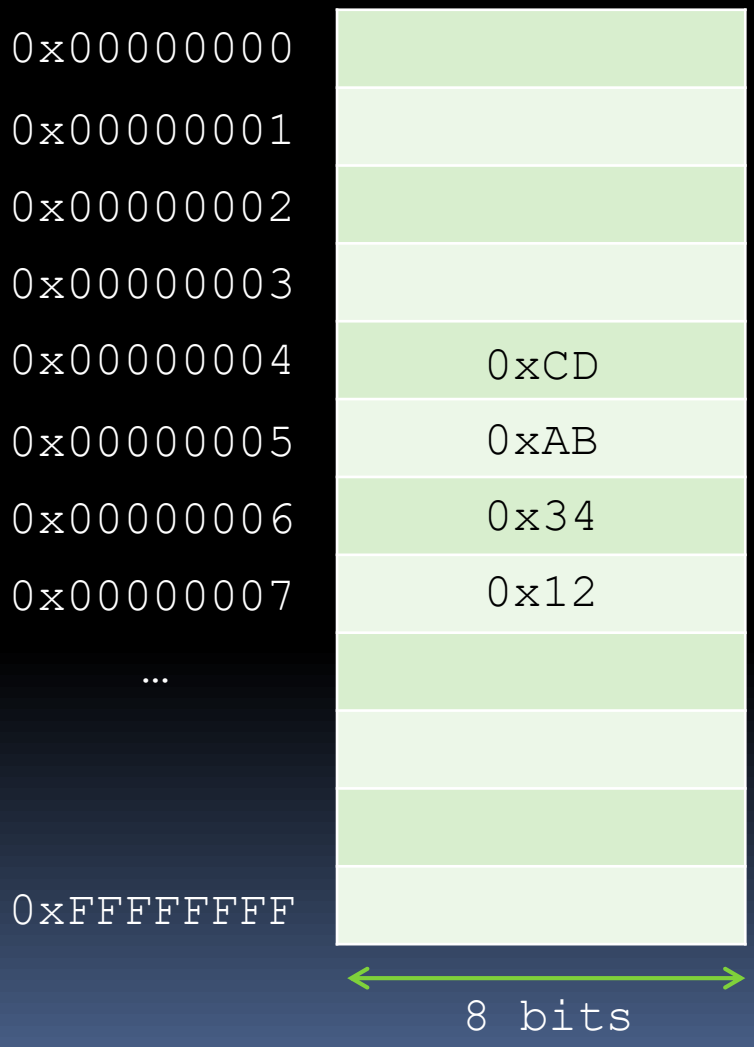



```
#assume $t0 contains
#0x00000004
sw $t1, 0($t0)
```

0x1234ABCD

32 bits

Little Endian Example




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

0x1234ABCD

32 bits

MIPS Endianness

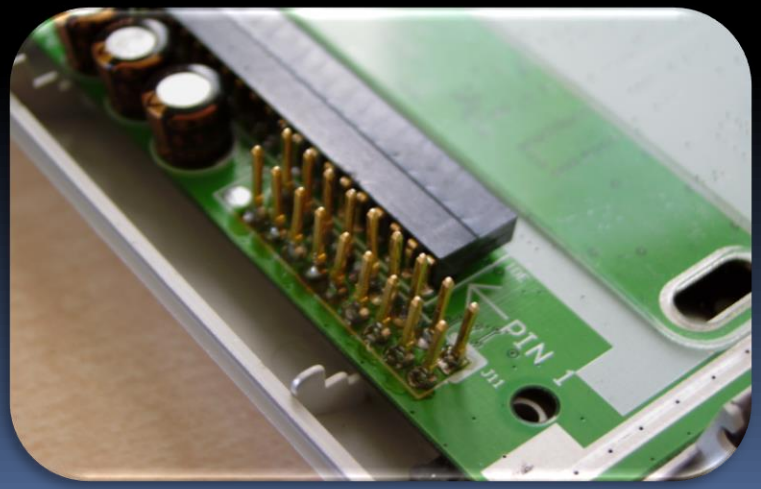
- MIPS processors are bi-endian, i.e., they can operate with either big-endian or little-endian byte order
- MARS simulator uses the same endianness as the machine it is running on.
  - X86 CPUs (like the one in my laptop) are little-endian, for instance.

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!

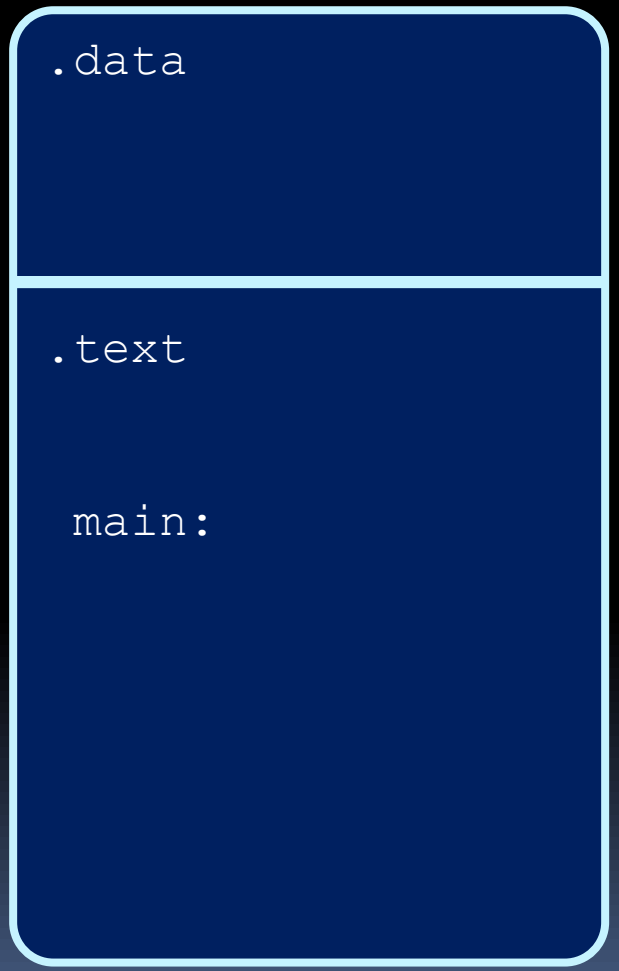
Instruction	Function	Syntax
`trap`	011010	i

- 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 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.



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 $at, immediate1`** **# load upper immediate**
    - The "**immediate1**" respresents the upper 16 bits of the memory address `label` corresponds to. These bits are loaded in the upper 16 bits of the dest. register. Lowest 16 bits are set to `0`.
    - Register `$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, 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:
```

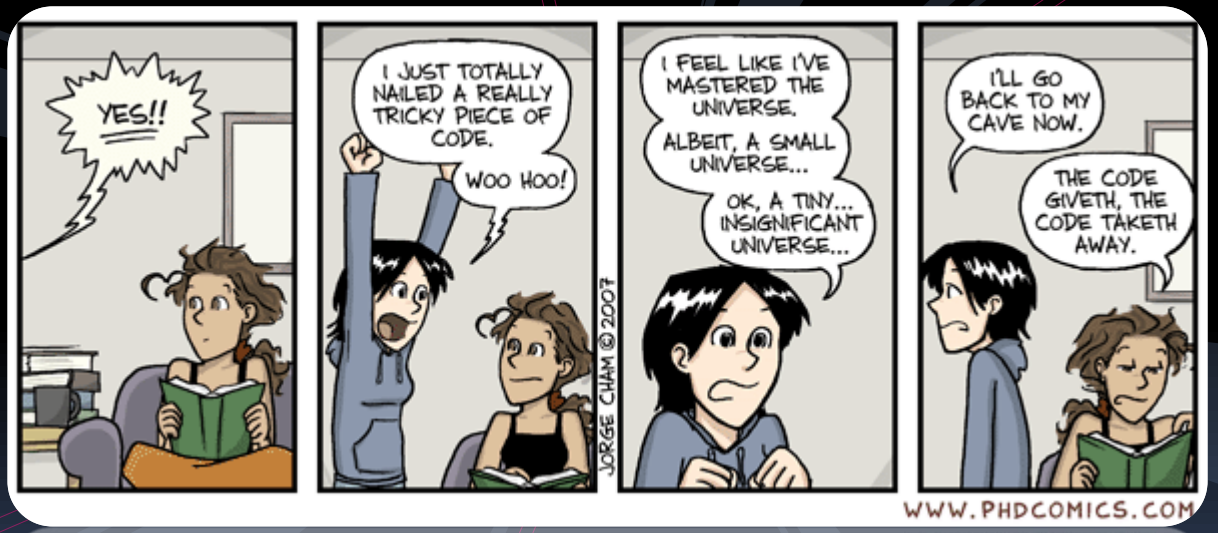
Example: A struct program

- 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

```
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       # $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

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

```
                .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
```

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)

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

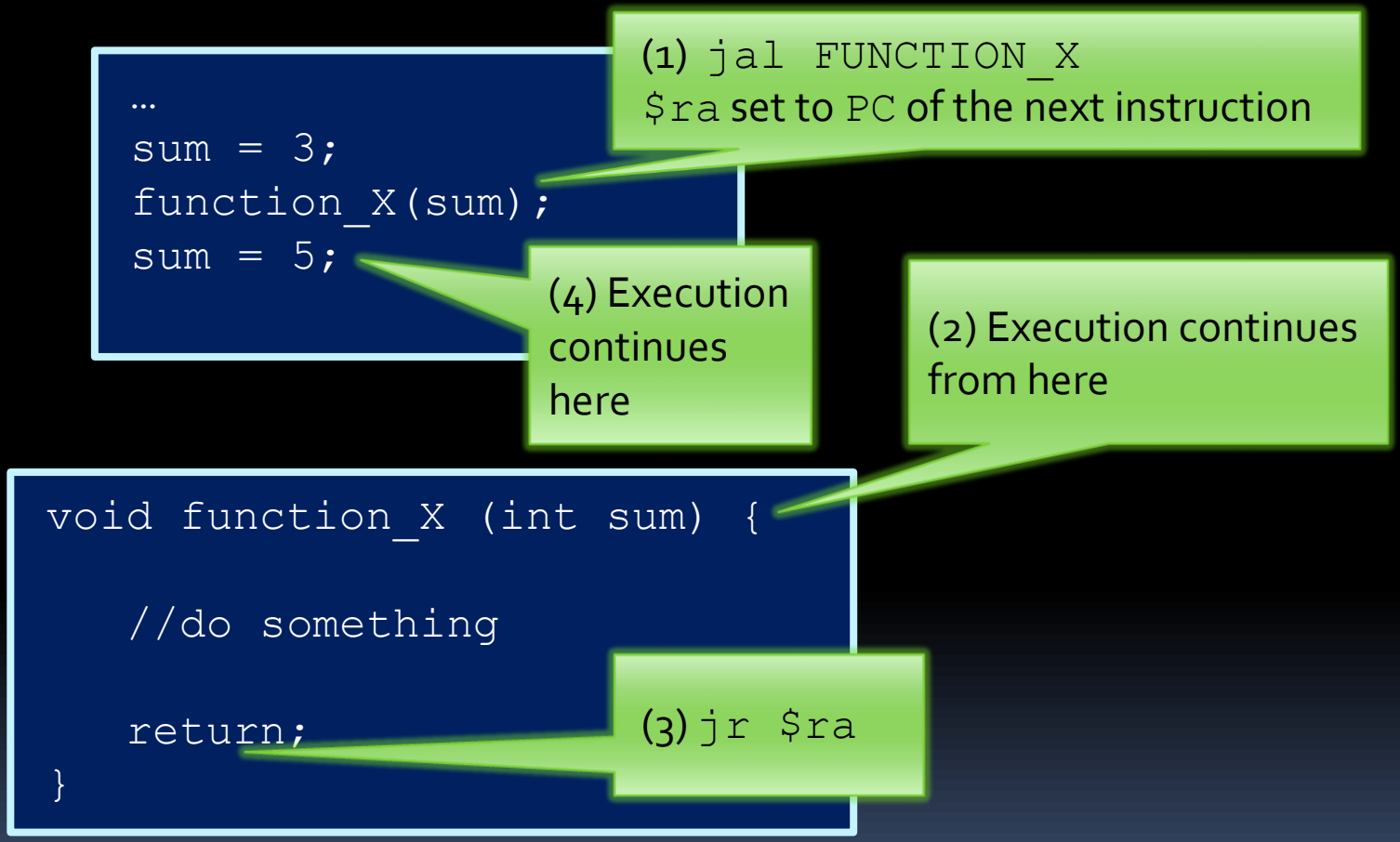
```
…
sum = 3;
function_X(sum);
sum = 5;
```

```
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) Execu…
`jr $ra`

Which `$ra`?
No way back! ☹

Solving the `$ra` dilemma

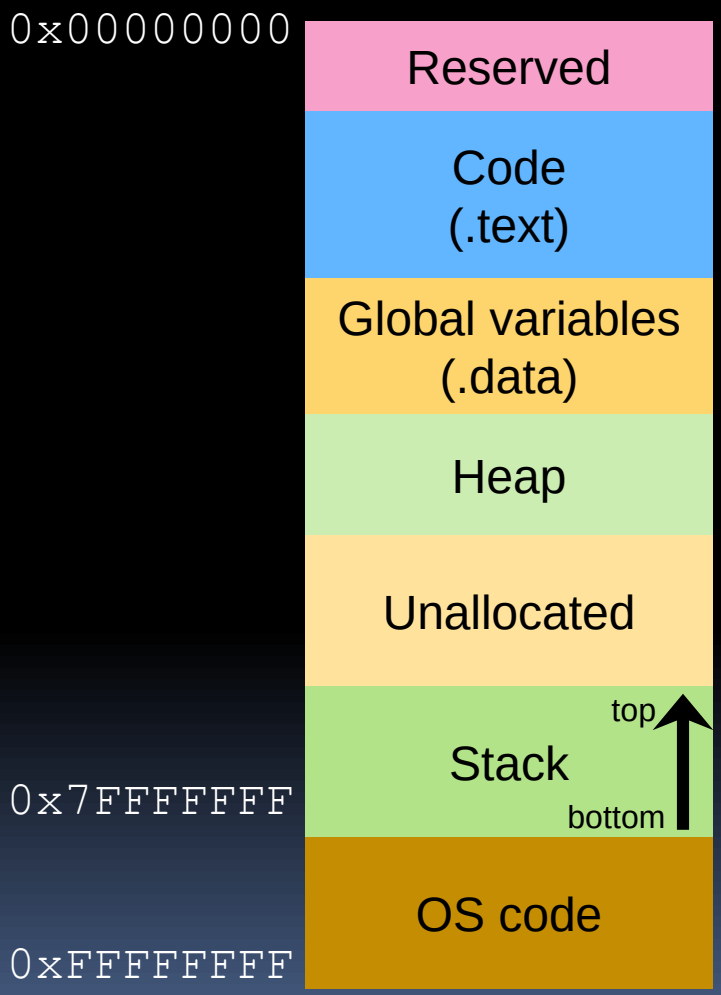
- How do we make sure that we don't clobber the value of `$ra` every time we call a nested function?
- Need to put `$ra` away somewhere for safe keeping if we know we're about to overwrite it.
- Would be ideal if the structure used for this made it easy to find the last item that was stored away.

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



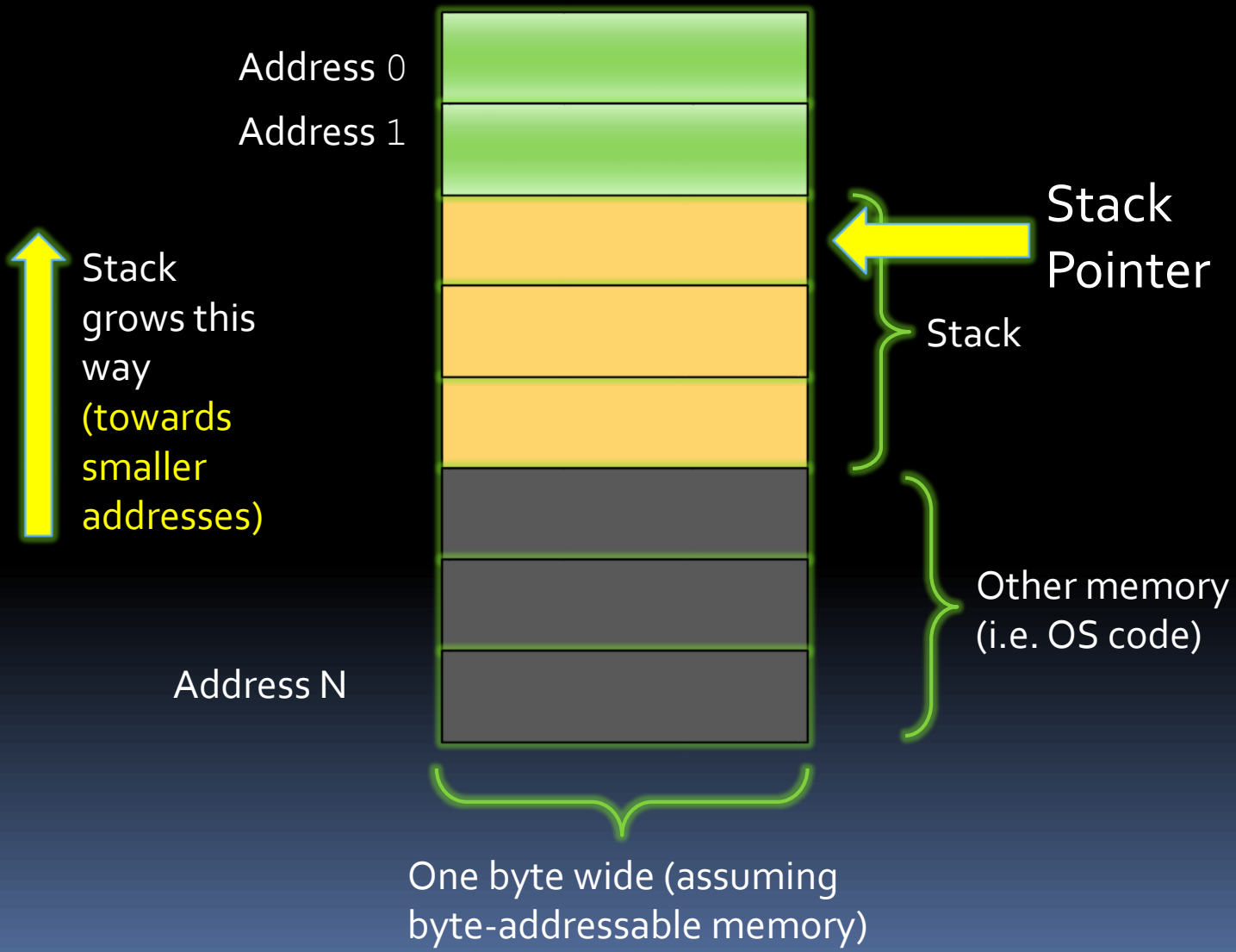
- The stack is a part of memory used for function calls etc.
- The stack grows towards smaller (lower) addresses
  - see arrow.
- The stack uses LIFO (last-in first-out) order.
  - Like a physical pile that you add and remove items from.

The stack to the rescue!

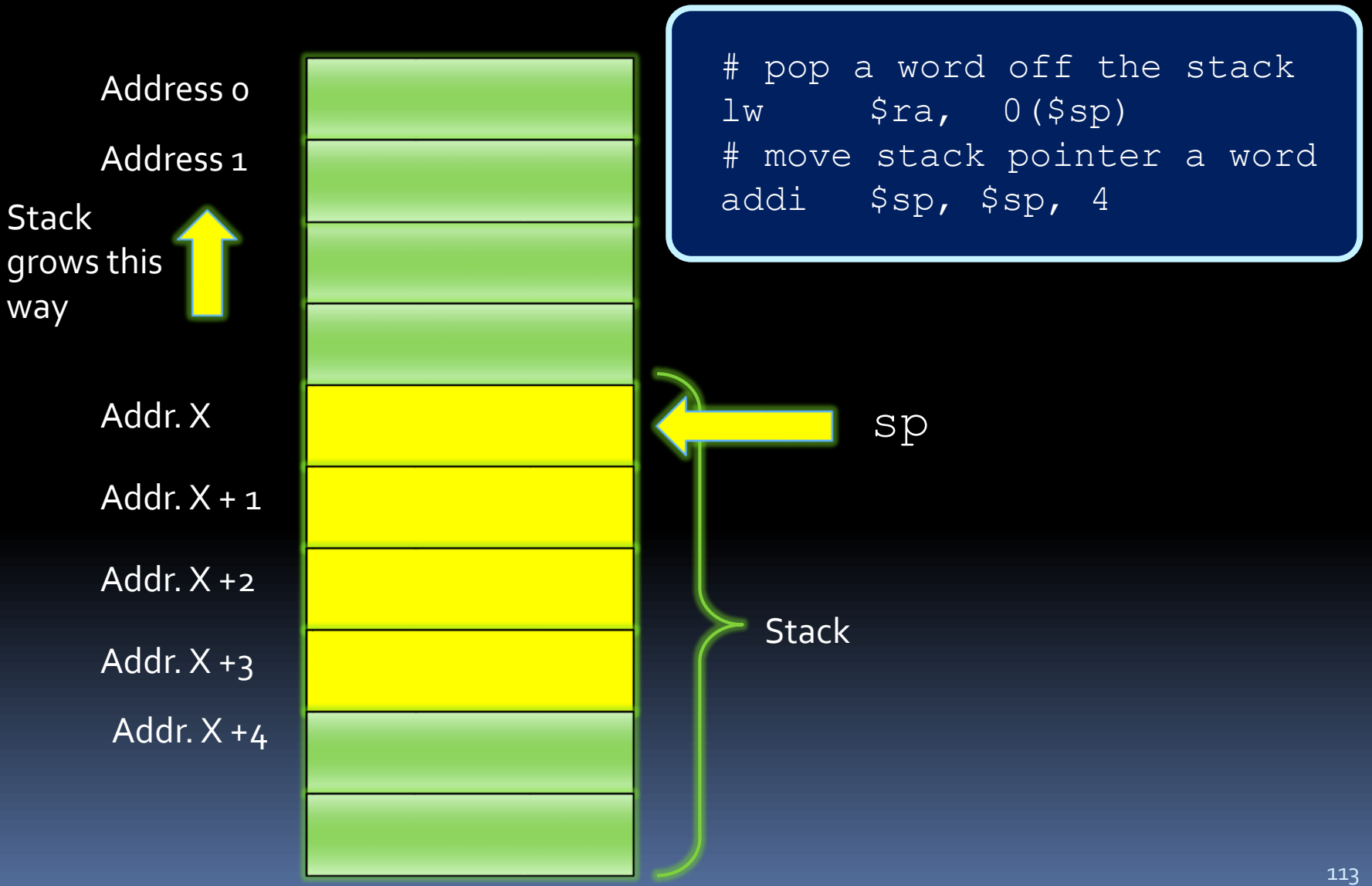
- When do you store values onto the stack?
  - Whenever you call a function and want to preserve values from getting overwritten (like `$ra`).
- What happens when you have nested function calls, each of which stores `$ra` on the stack?
  - Different `$ra` values will exist in layers on the stack over time.

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

The Stack, illustrated

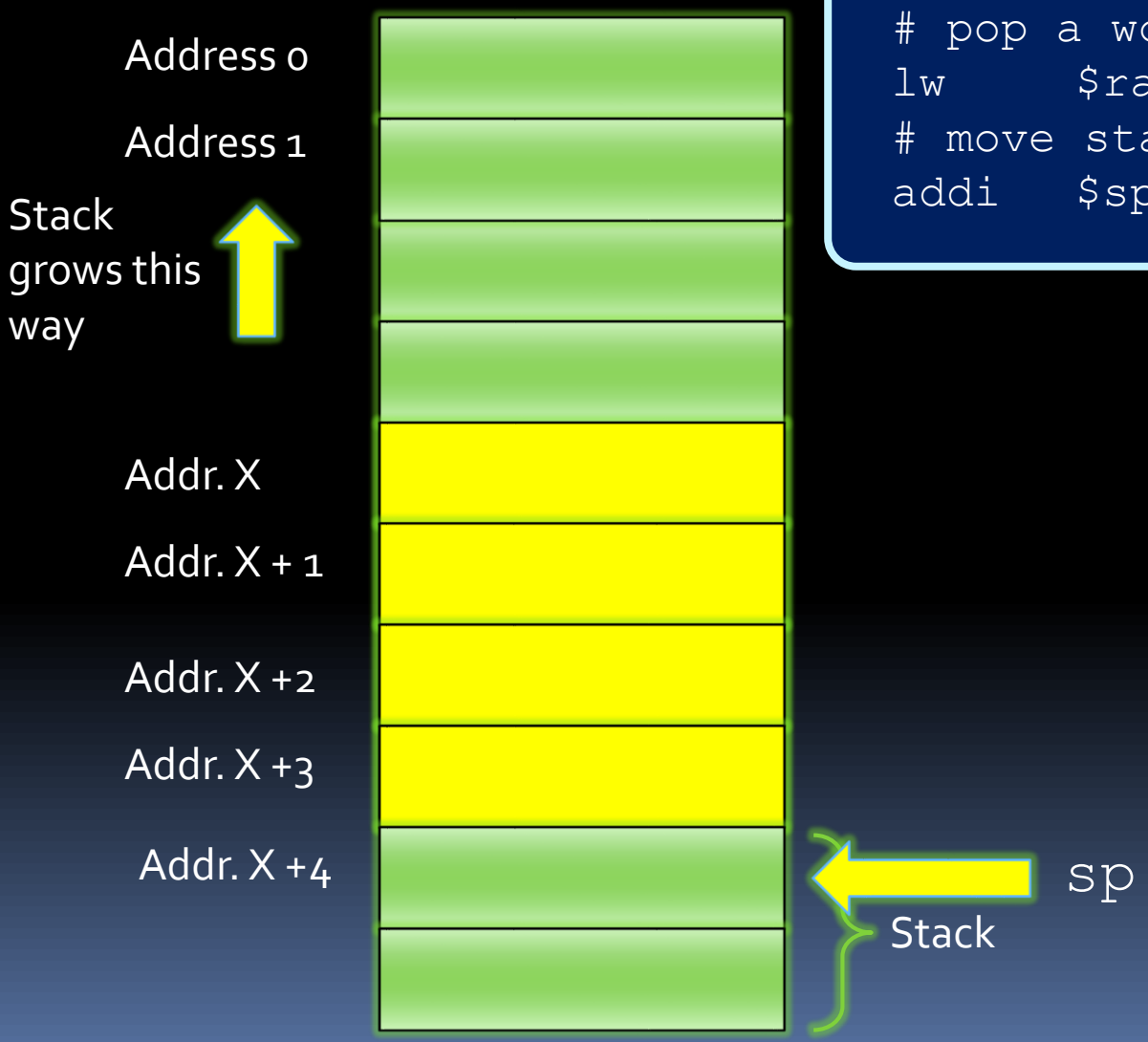


Popping Values off the stack - Before




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

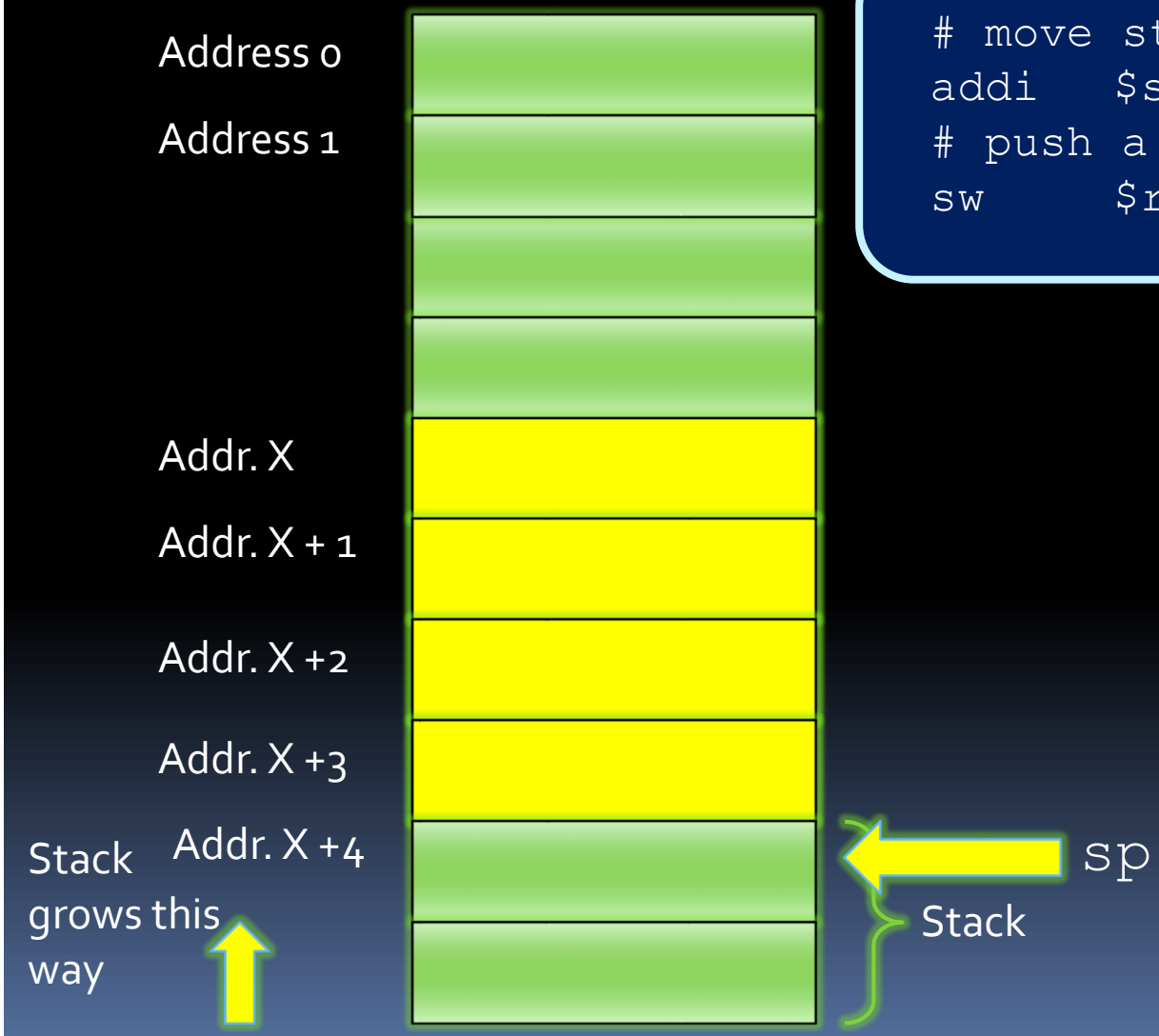
Popping Values off the stack - After




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

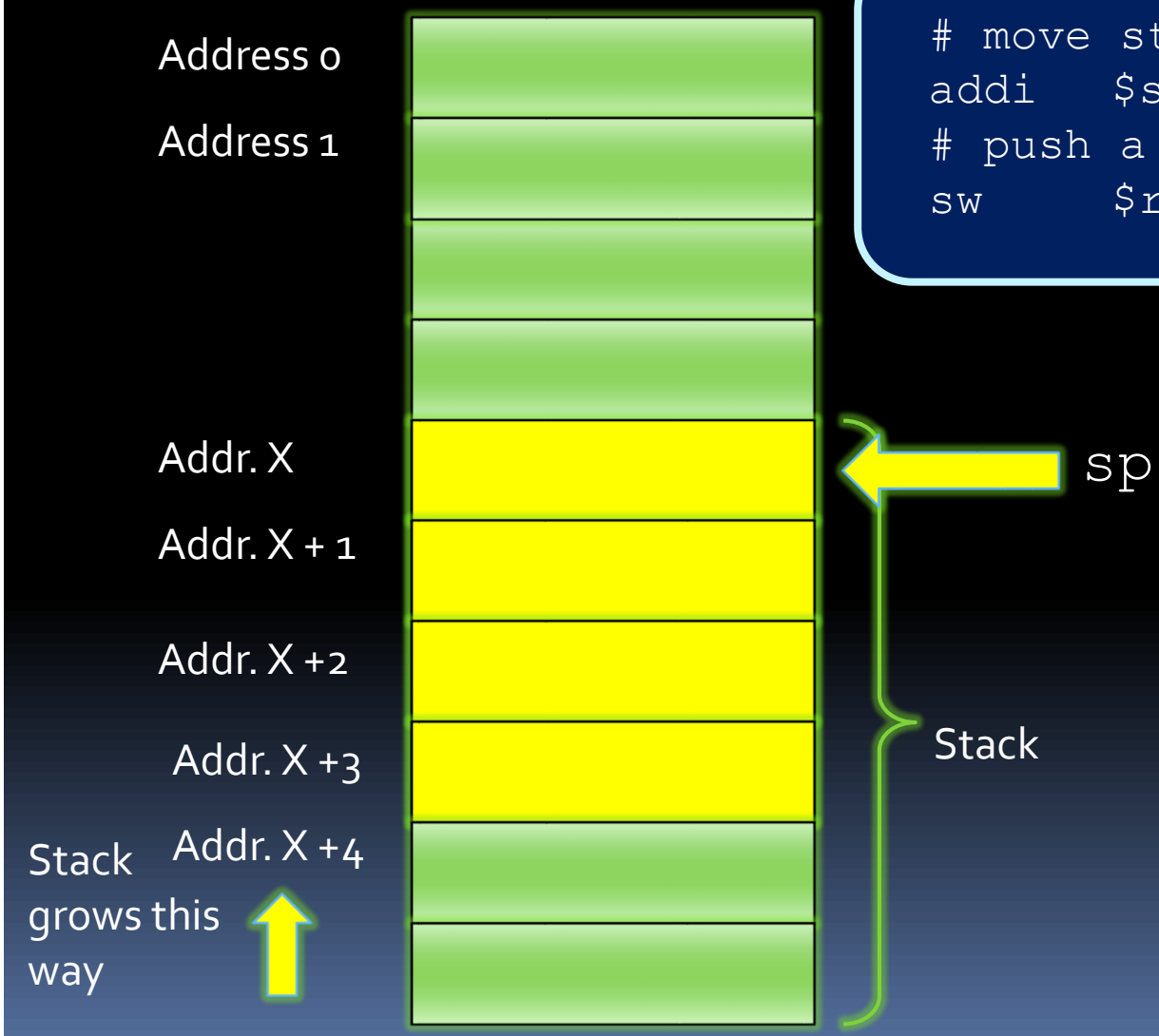
Pushing Values to the stack - Before

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



Pushing Values to the stack - After

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



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 st

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

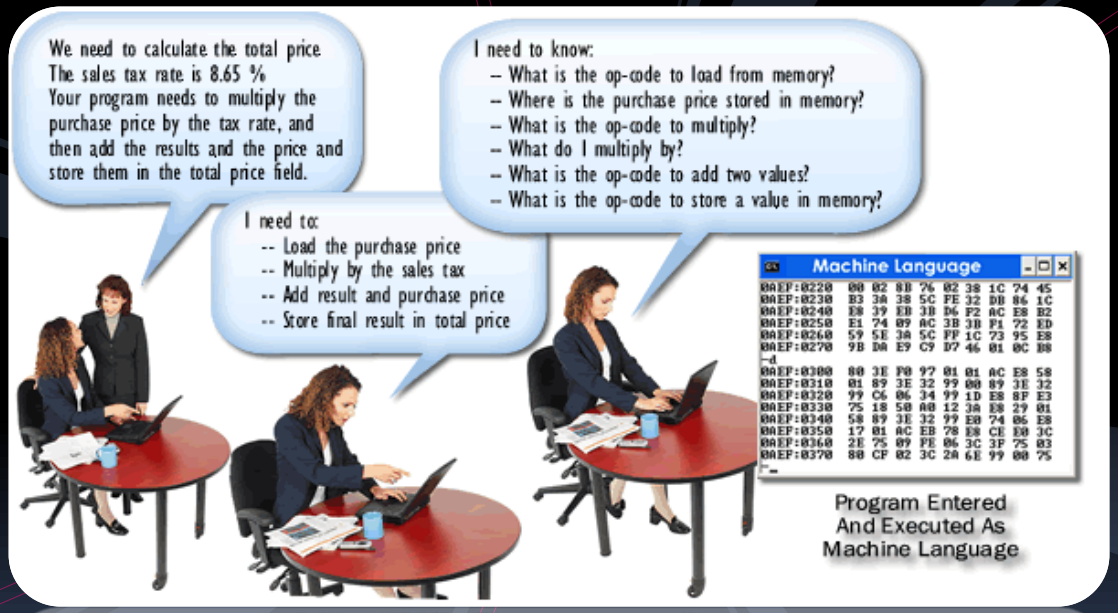
initialization

main algorithm

end

```
strcpy:  lw    $a0, 0($sp)          # pop y address
         addi  $sp, $sp, 4          #  off the stack
         lw    $a1, 0($sp)          # pop x address
         addi  $sp, $sp, 4          #  off the stack
         add   $t0, $zero, $zero    # $t0 = offset i
L1:      add   $t1, $t0, $a0        # $t1 = y + i
         lb    $t2, 0($t1)          # $t2 = y[i]
         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
         addi  $t0, $zero, 1        #  the top of
         sw    $t0, 0($sp)          #  the stack
         jr    $ra                  # return
```

Function Considerations



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

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!

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 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.
  - 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

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

- 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.

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 `$t0` != `0`,
  - Calculate `n-1`
  - Store (i.e. push) `$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

Translated recur

```
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
```

```
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
```

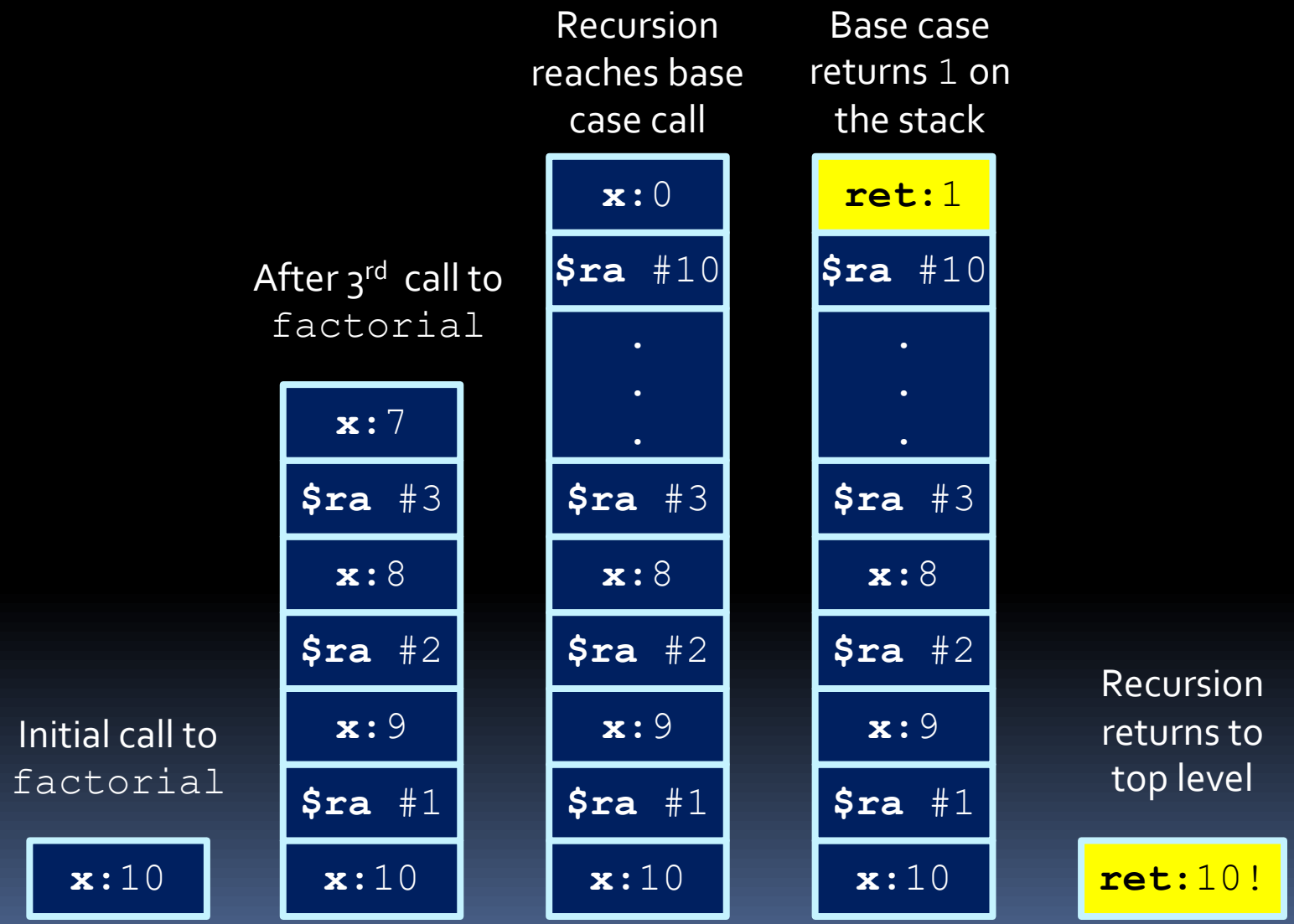
Translated recur

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

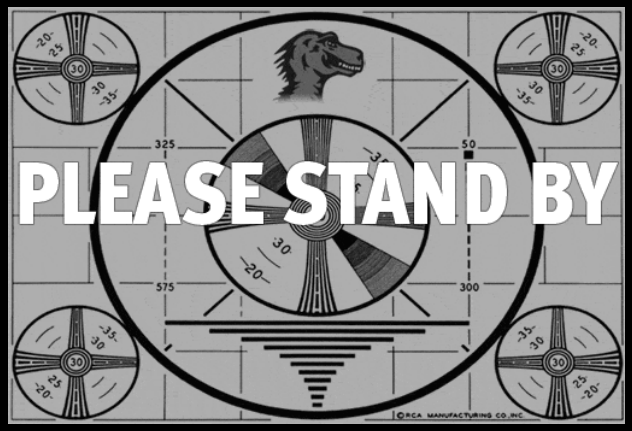
```
(continued from part 1 – returning from recursive call)
            lw     $t2, 0($sp)          # pop return value
            addi   $sp, $sp, 4          #   from stack
            lw     $ra, 0($sp)          # restore return
            addi   $sp, $sp, 4          #   address value
            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.
  - Vectored 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