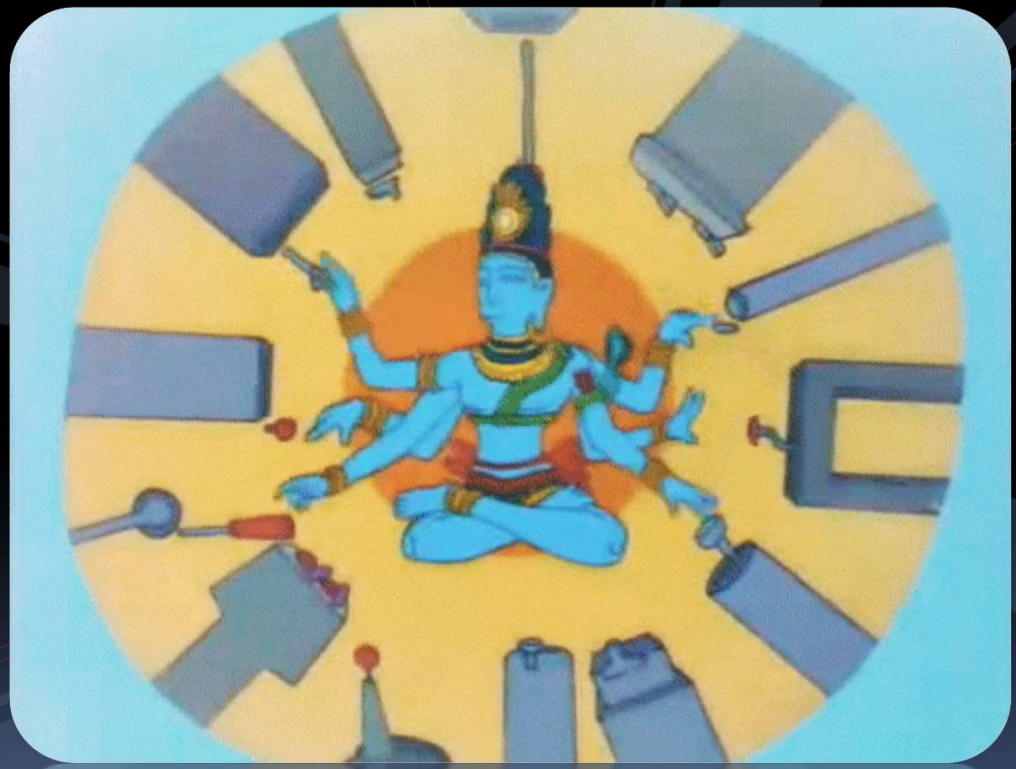
Assembly Language

* Created with contributions by Myrto Papadopoulou and Frank Plavec.

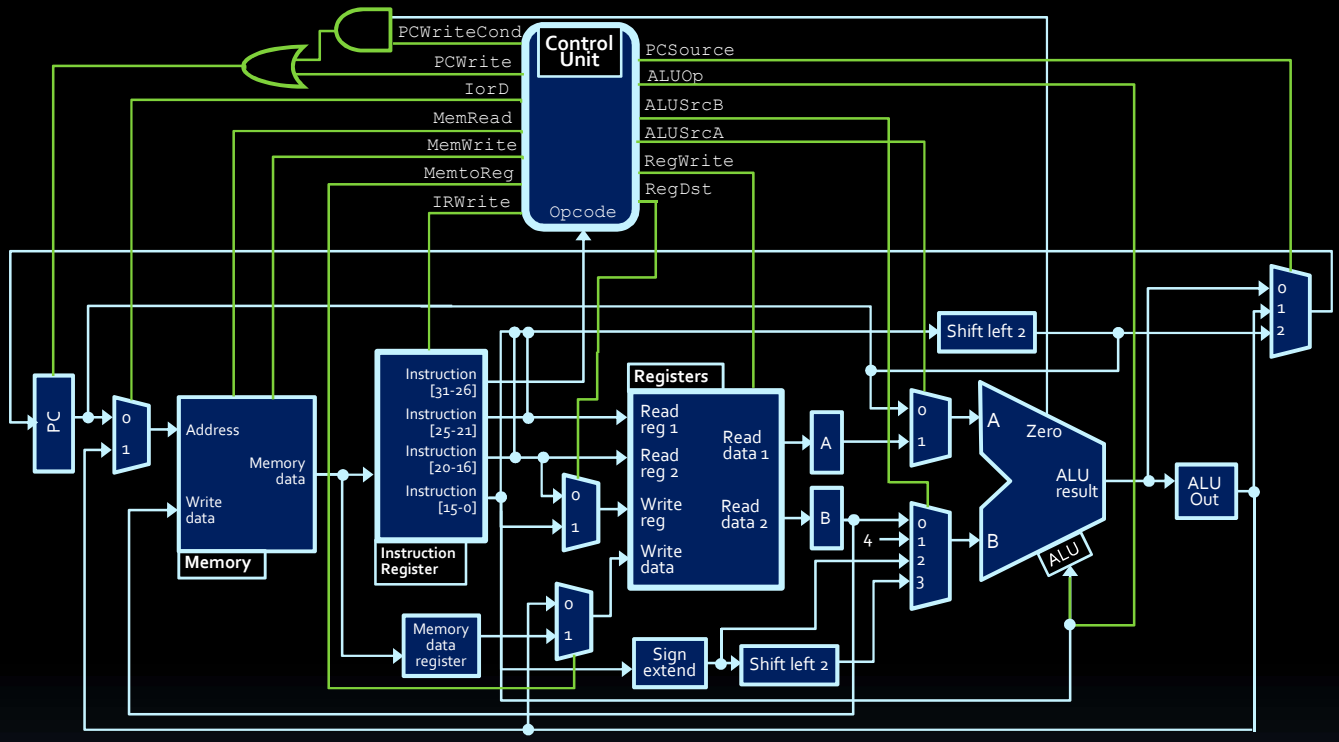
Programming the processor

- 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



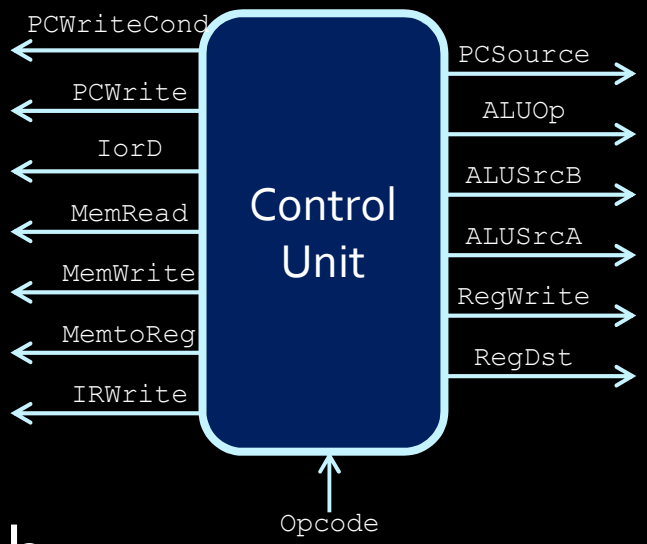
MIPS Datapath



- 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

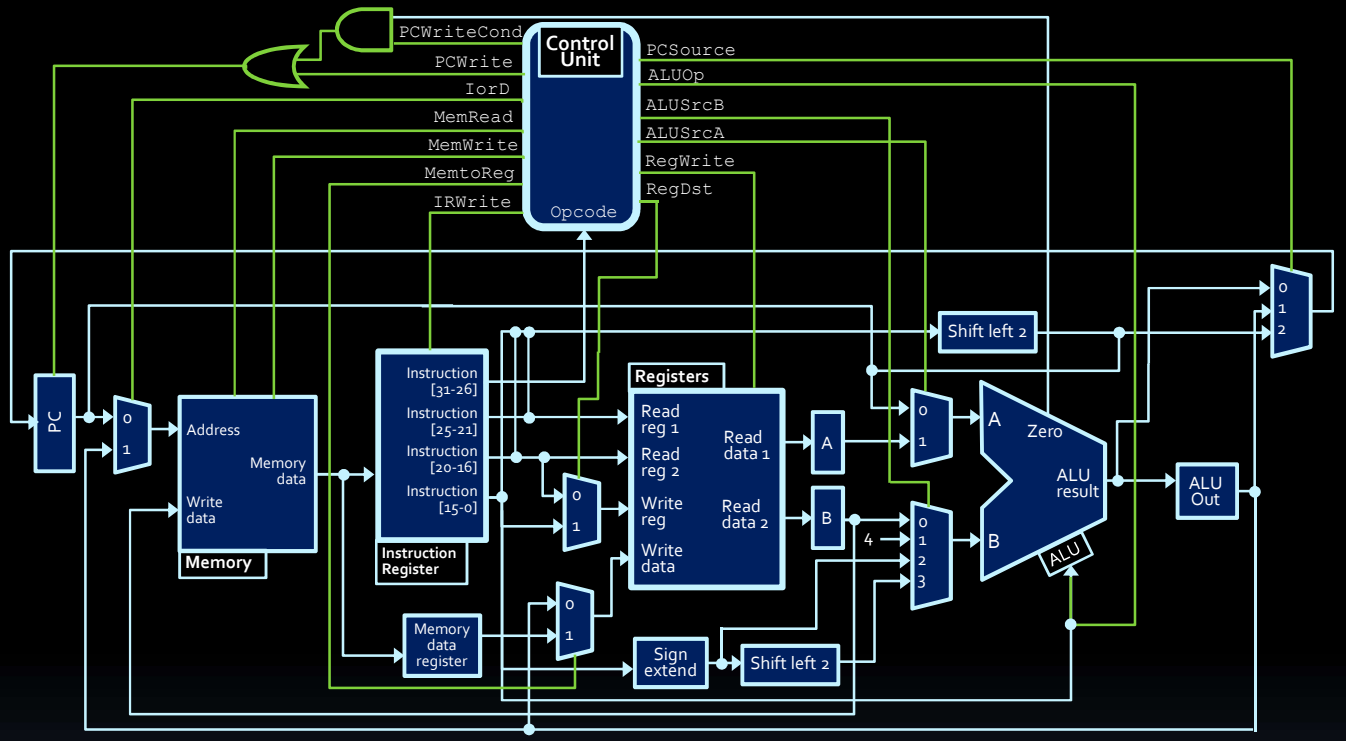
- Need to understand the role of each signal, and what value they need to have in order to perform the given operation.
- So, what's the best approach to make this happen?



Basic approach to datapath

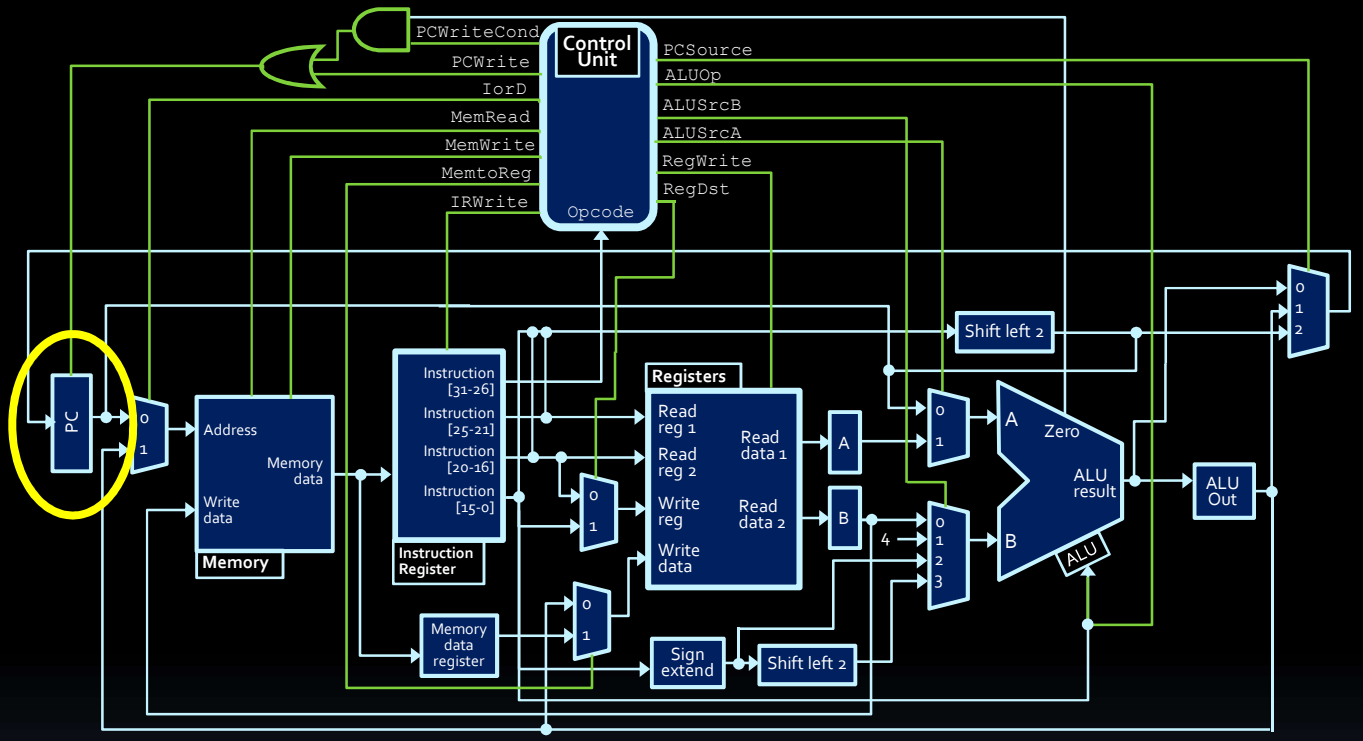
1. Figure out the data source(s) and destination.
2. Determine the path of the data.
3. Deduce the signal values that cause this path:
   a) Start with `Read` & `Write` signals (at most one can be high at a time).
   b) Then, mux signals along the data path.
   c) Non-essential signals get an `X` value.

Example #1: Incrementing PC



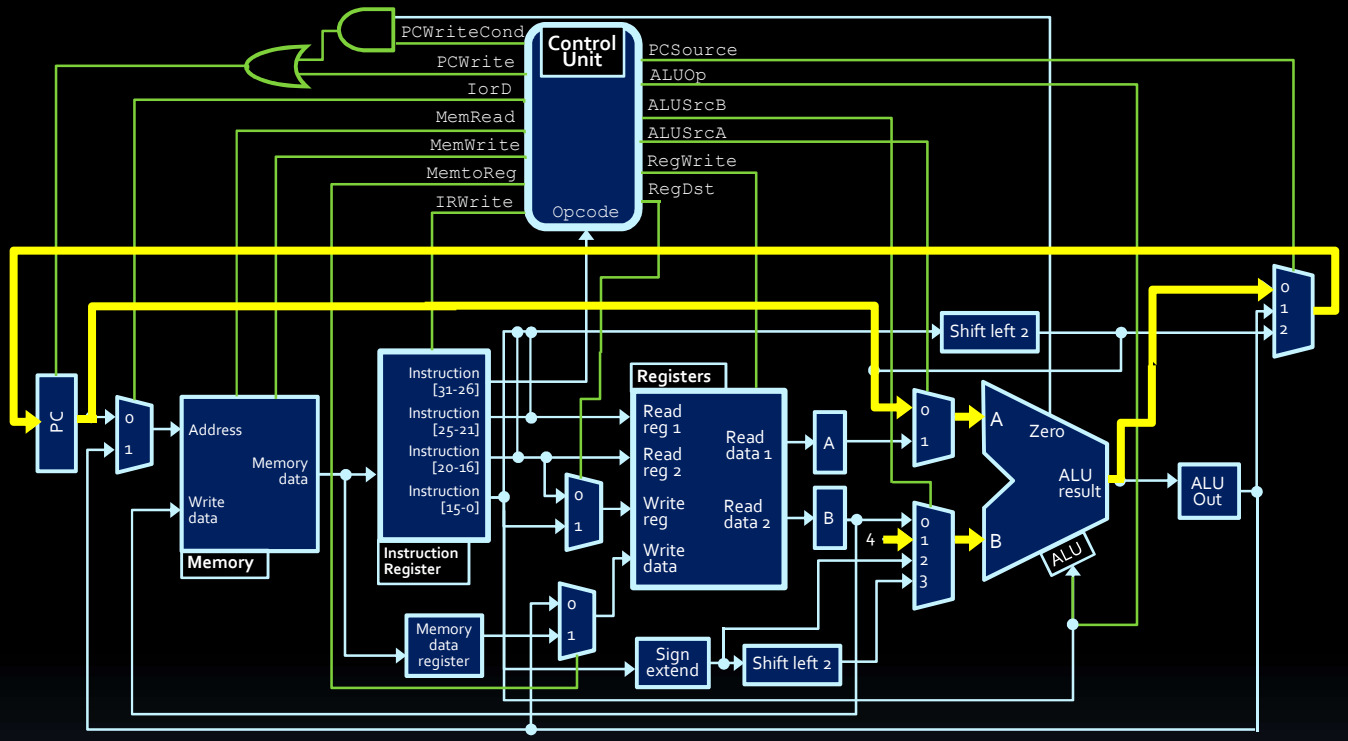
- Given the datapath above, what signals would the control unit turn on and off to increment the program counter by 4?

Example #1: Incrementing PC



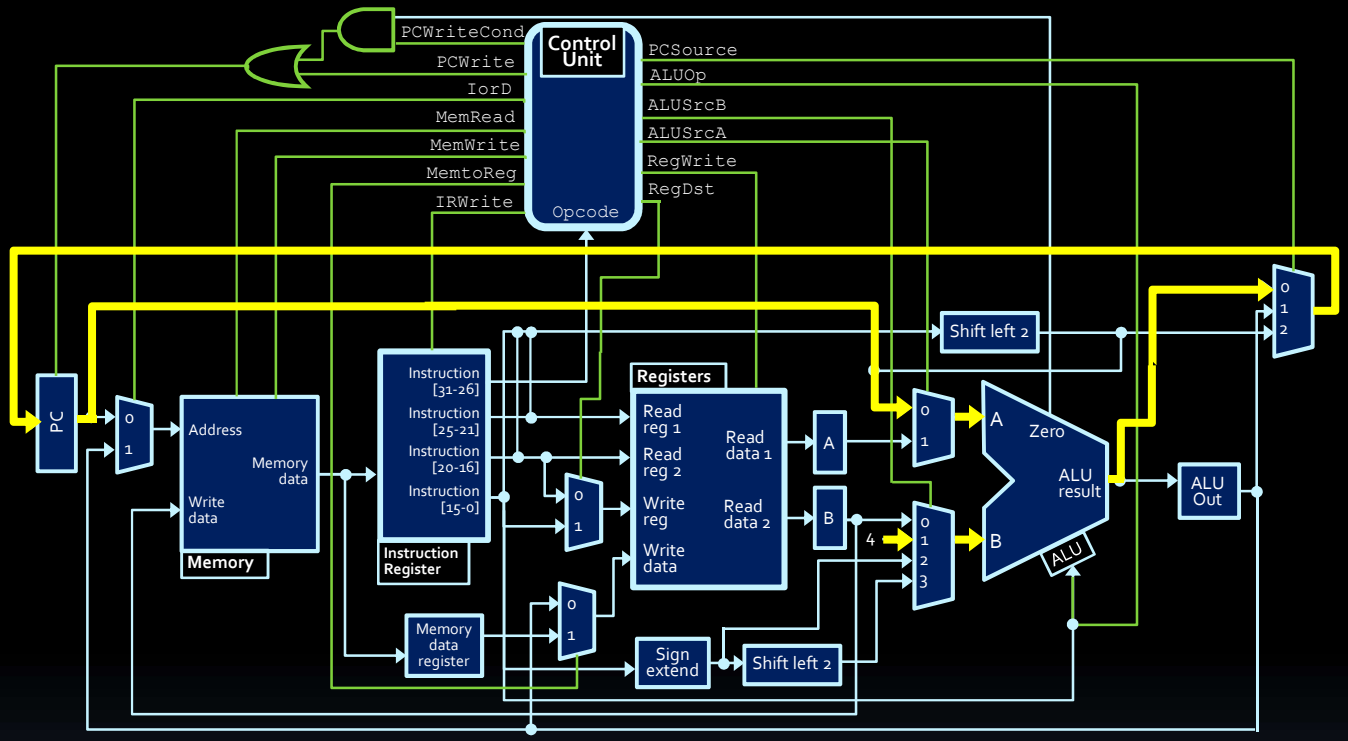
- Step #1: Determine data source and destination.
  - Program counter provides source,
  - Program counter is also destination.

Example #1: Incrementing PC



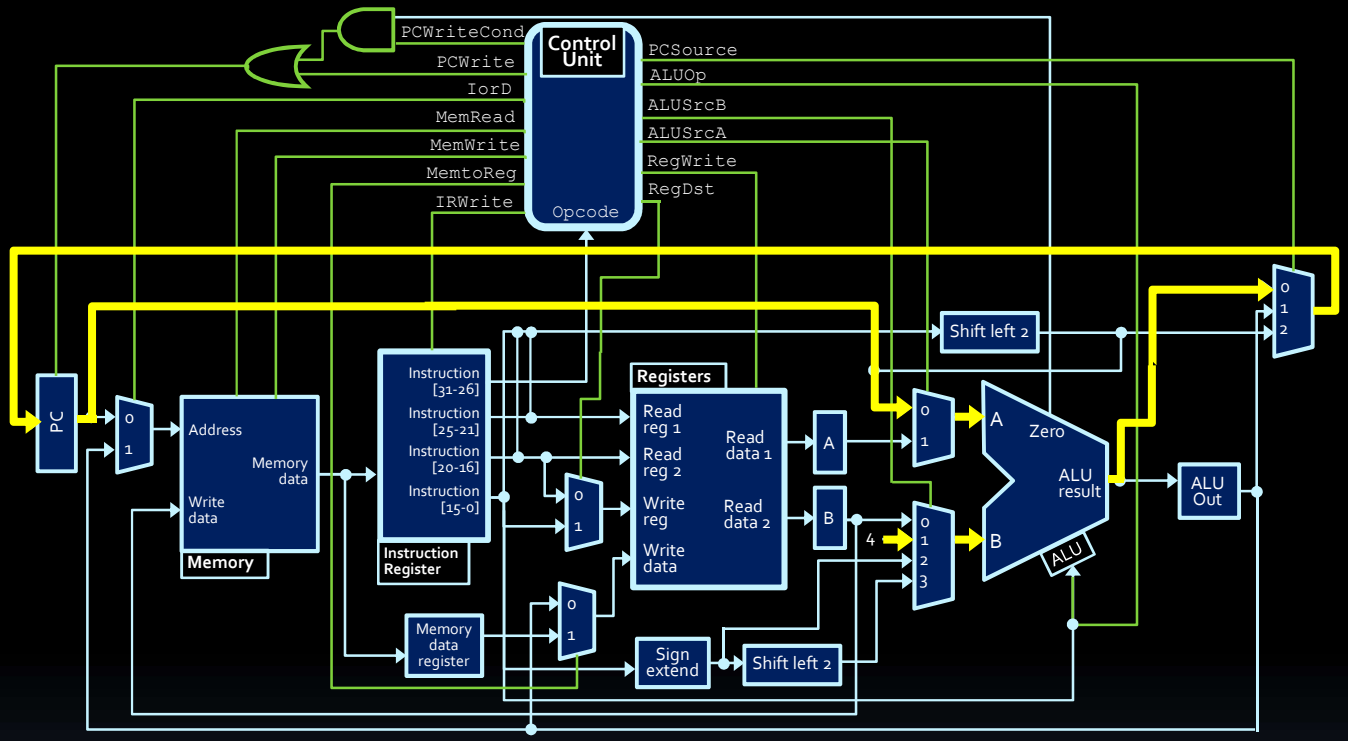
- Step #2: Determine path for data
  - Operand A for ALU: Program counter
  - Operand B for ALU: Literal value 4
  - Destination path: Through mux, back to PC

Example #1: Incrementing PC



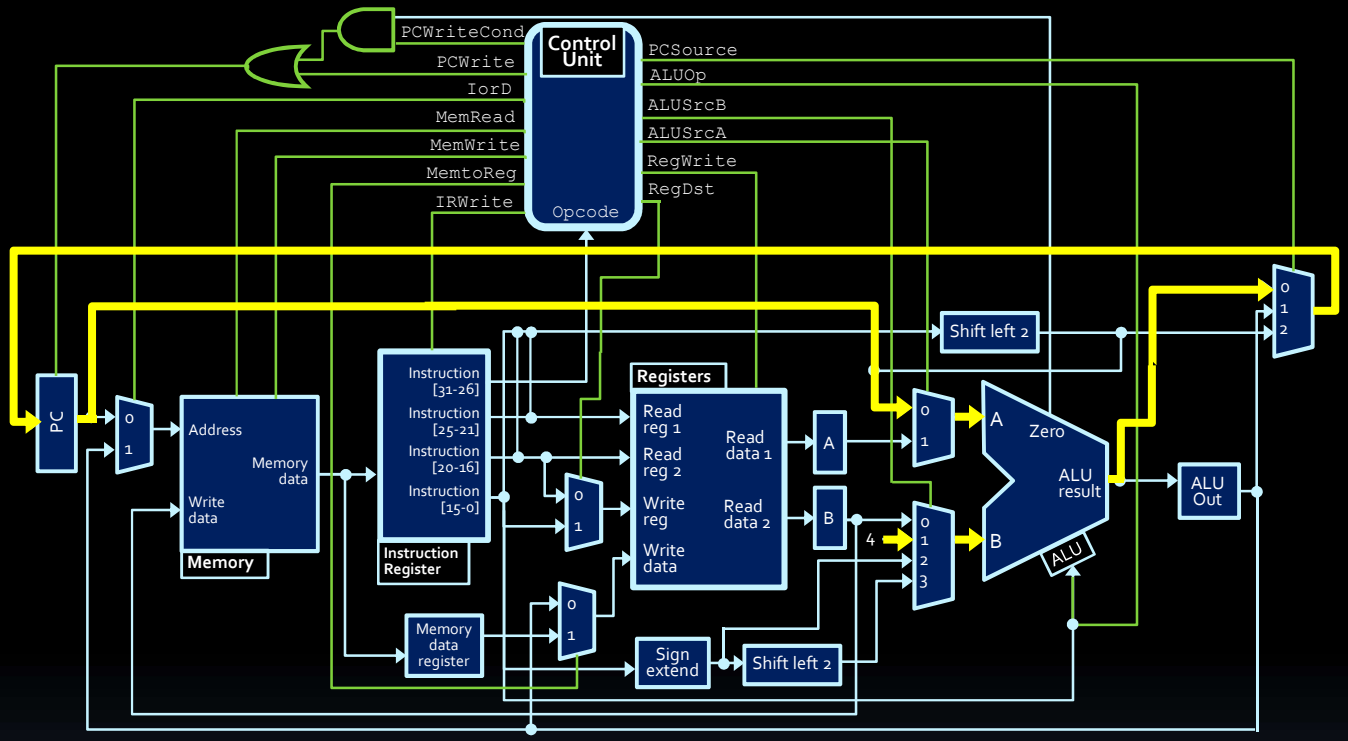
- Setting signals for this datapath:
  1. Read & Write signals:
     - `PCWrite` is high, all others are low.

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)

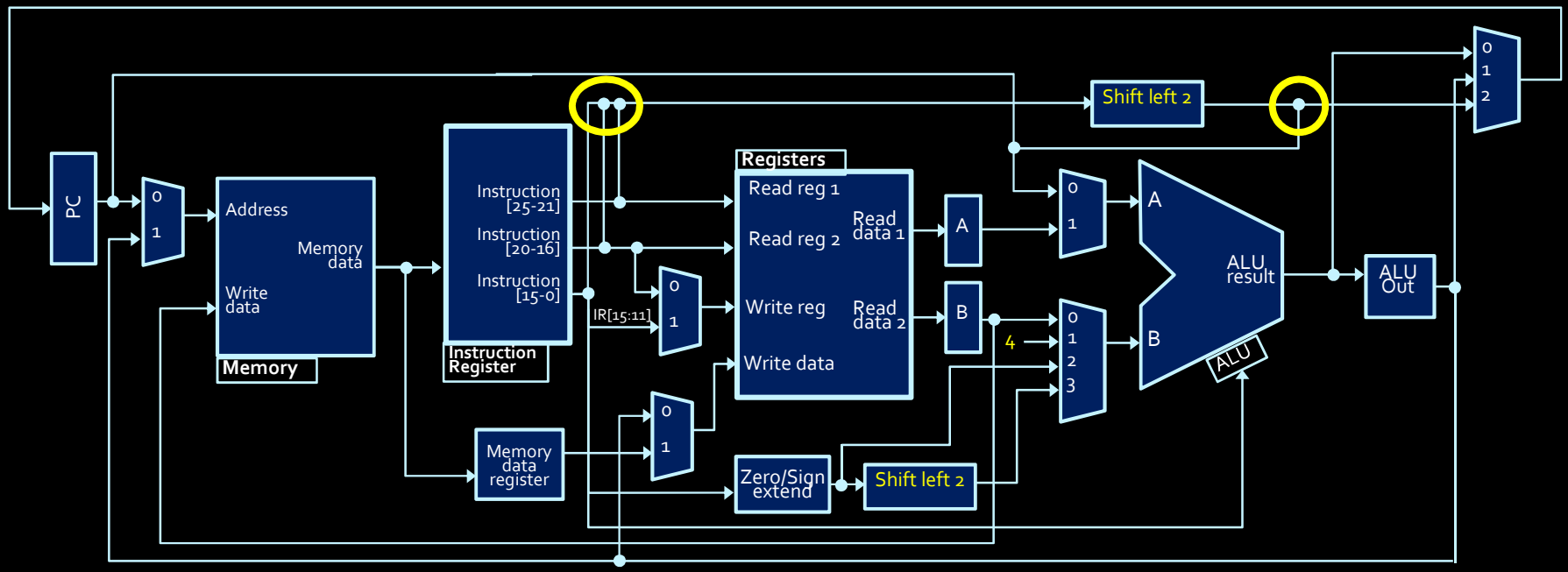
- PCWrite = 1
- PCWriteCond = X
- IorD = X
- MemRead = 0
- MemWrite = 0
- MemToReg = X
- IRWrite = 0
- PCSource = 0
- ALUOp = 001
- ALUSrcA = 0
- ALUSrcB = 01
- RegWrite = 0
- RegDst = X

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.

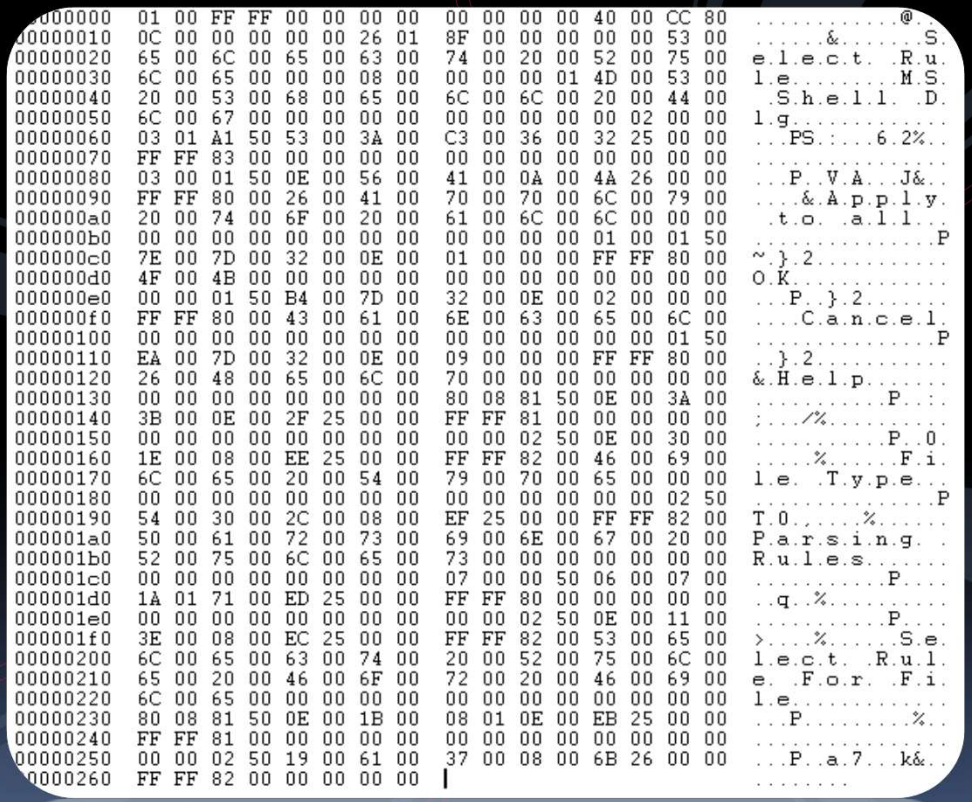


The remaining 26 bits

- 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



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
```

adds t1 and t2 into t3

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

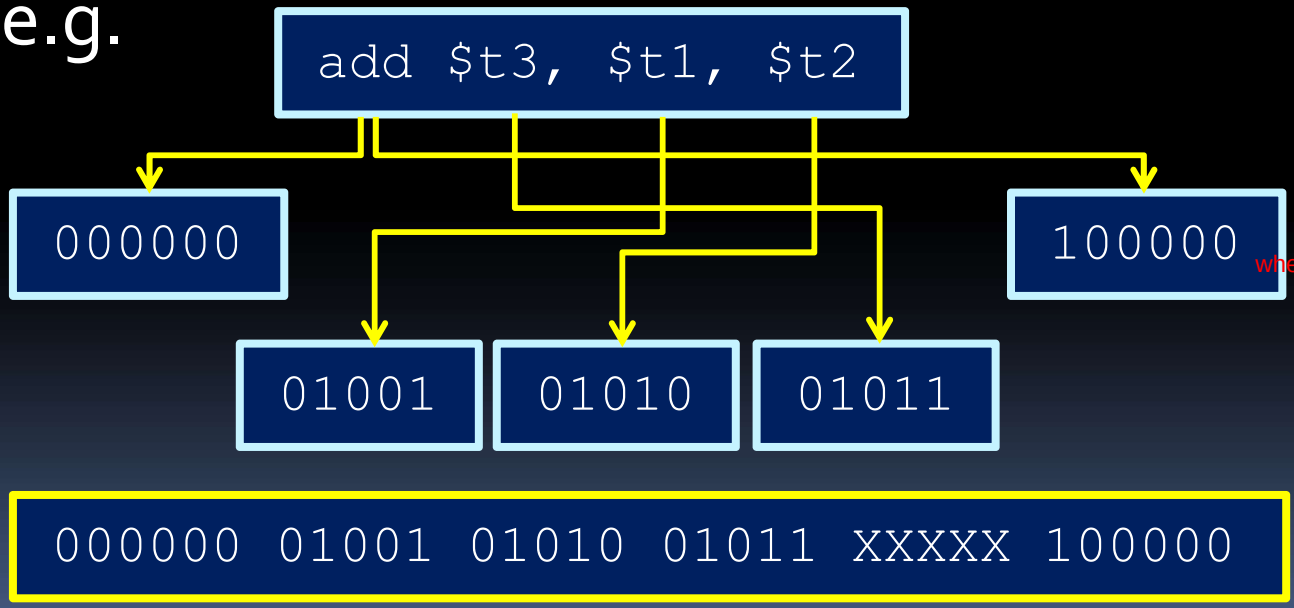
in final will give one of these and you need to give corresponding one

  - Machine code instruction:

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

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

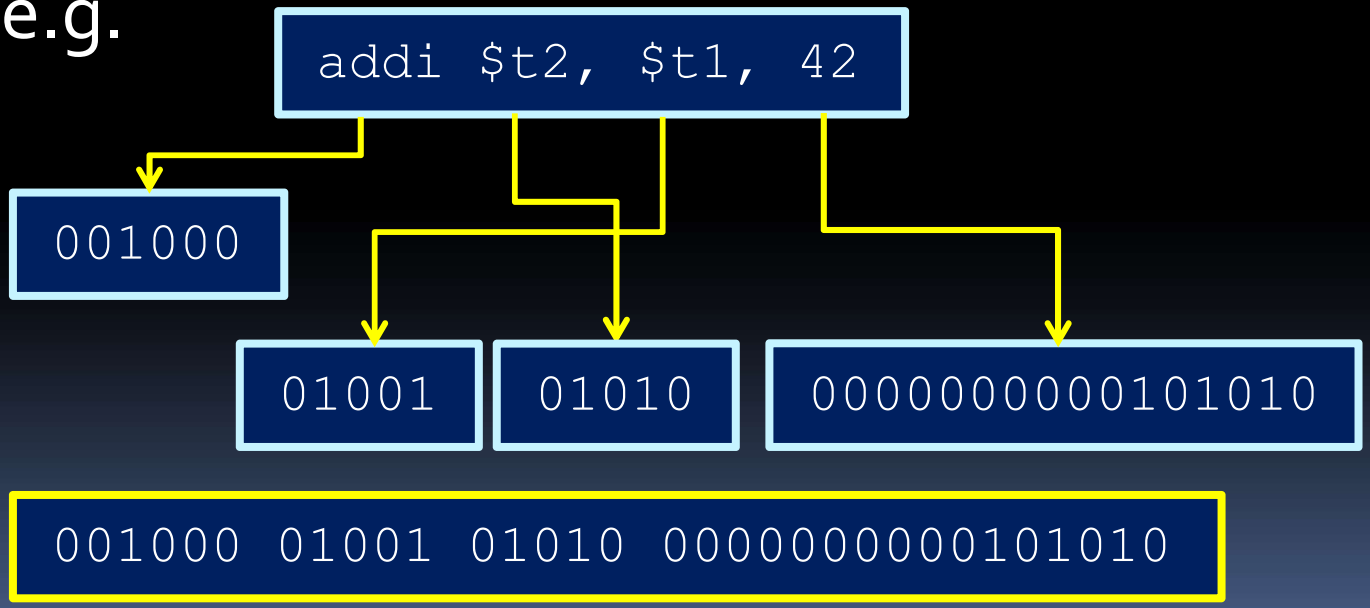
- The `add` instruction is one of many operations that processes two register values and stores the result in a third.
  - This is called an R-type instruction.
  - Any operations whose inputs and outputs are all registers are called R-type, even if they involve less than three registers.
    - e.g. 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` (`$vo`, `$v1`): return values
    - Registers `4-7` (`$ao-$a3`): function arguments
  - Some are used by programs to store values:
    - Registers `8-15`, `24-25` (`$to-$t9`): temporaries
    - Registers `16-23` (`$so-$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.

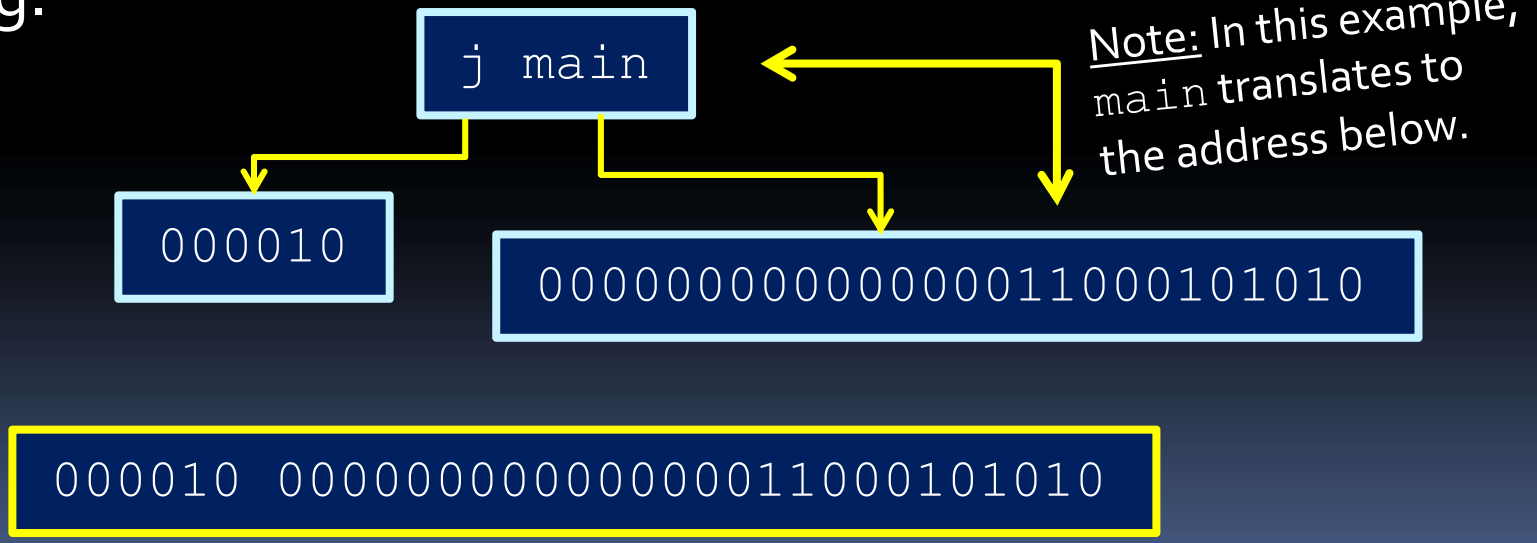
I-type instructions

- 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.



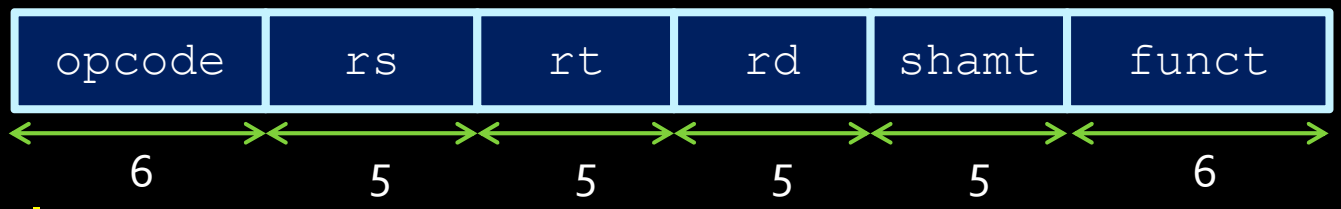
J-type instructions

- 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:

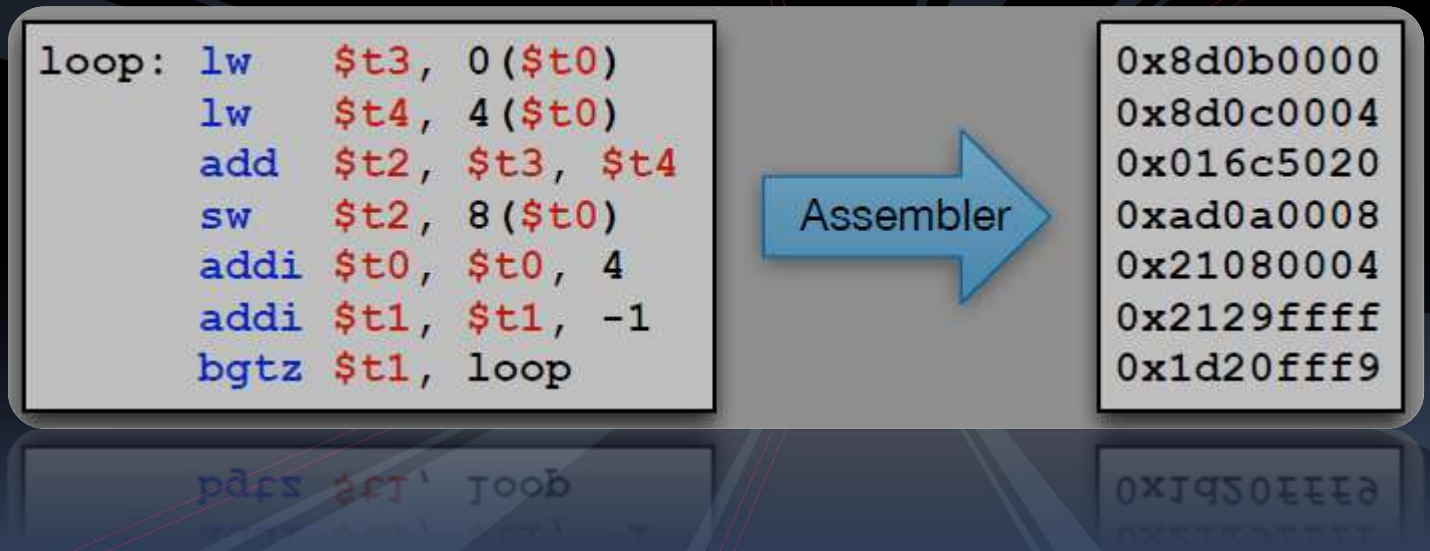
- J-type:

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).

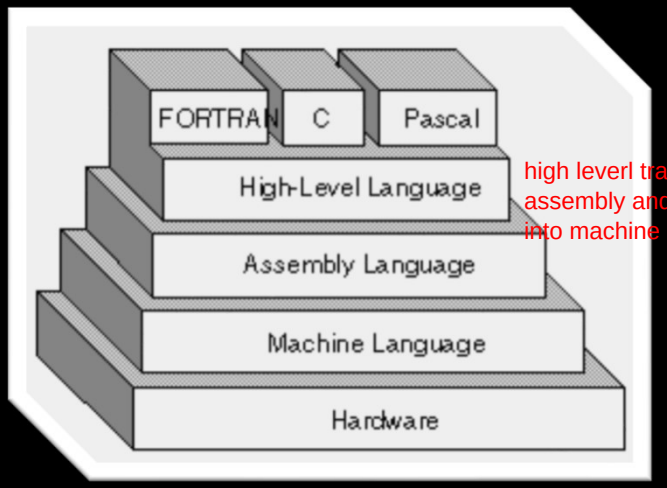
Assembly Language Overview

back here now for the start of the actual lecture (Nov 21)
a better way to make a instruction rather than program all as 32 bit numbers



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!



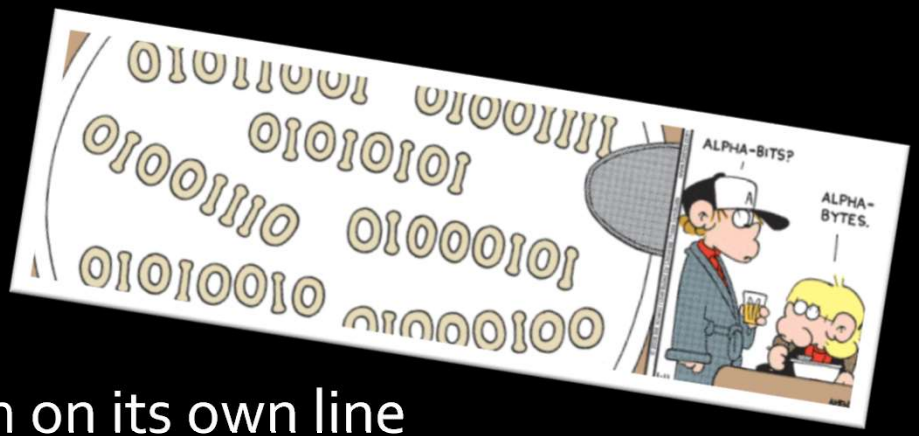
high leverl translated to assembly and 1-1 tranalation into machine

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.

RISC - Reduced instrution set -> fewer instructions to learn but havet o learn how to combine them to do cetrain things you think that instrcutions should exist for

MIPS Instructions



- Things to note about MIPS instructions:
  - Instruction are written `<instr> <parameters>`
    eg add
  - 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 ALU opps	`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%

pull stuff from mem and do something and then put back

3 major types of instructiions

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

branches and jumps instructions allow you to go back to another step and redo something eg think about loops in high level lanaguges
ALu opps instrcutions
data transfer instructions

Assembly Language Instructions

Arithmetic instructions

addi $t3, $zero, -1
mult $t0, $t3
need these two to get mult i
need to add the num and then mult

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

add vs addi

unsigned

i is nor a reg since no dollar sign in front of it

Sign extend i and add it, i = immediate - i type instruction - i value is the other source in the i instruction eg incrementing by 1, or add 15, one of the operands is not a variable is a literal ie an actual value

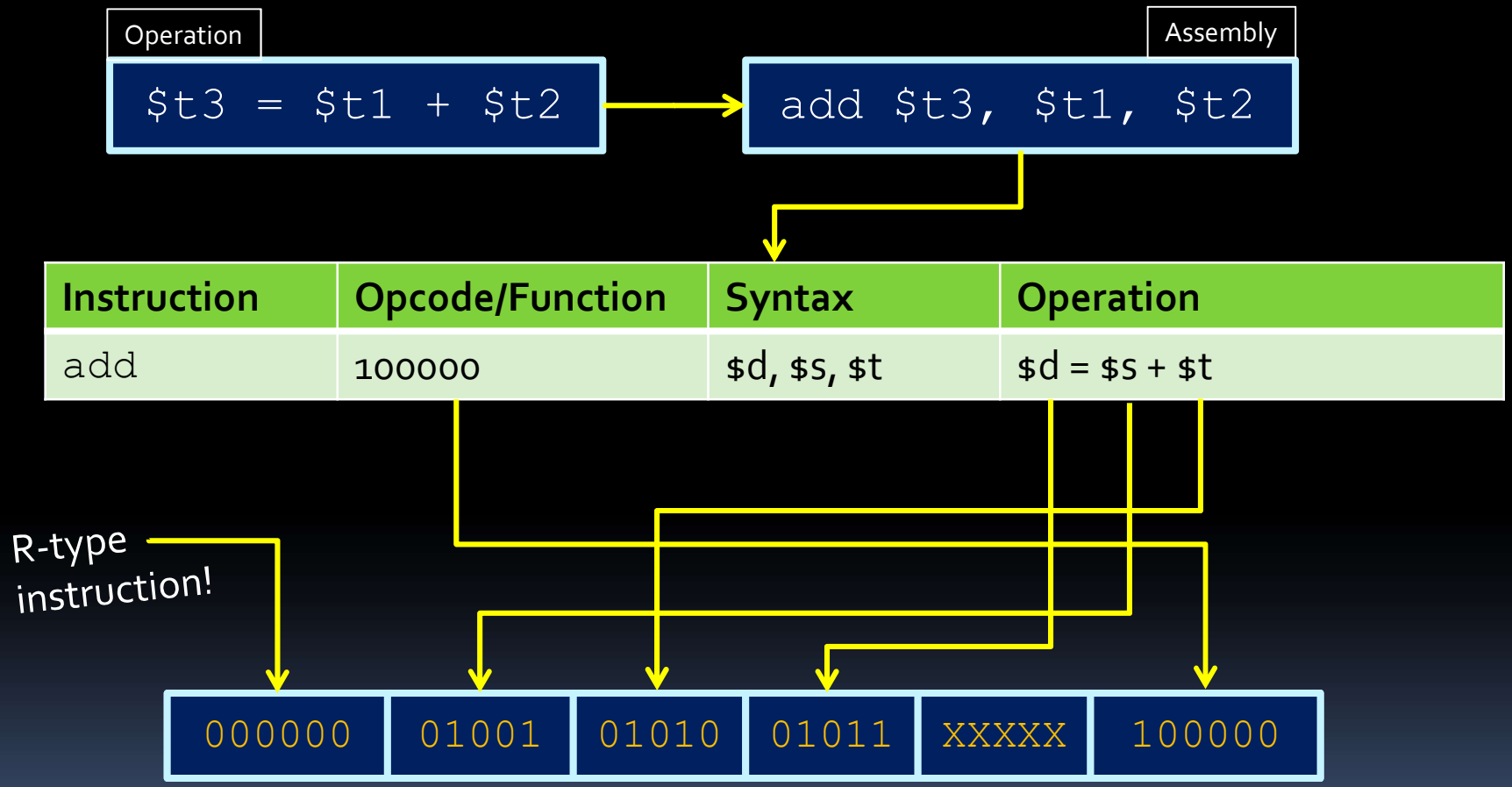
add immediate unsigned
is unsigned just add 1 to the largest + number
but if signed we will go from most pos to most neg if add 1

need to store due to size eg 32 x32 bit = 64 bit num
num to big to store in regs
so there is a special reg that is 64 bytes and can take higher or lower half and look at seperate - hi and lo
when you divide
lo stores quoitent and hi stores remainder

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

example $t0, $t1, $t2, we want to subtract t1 from t0 and store in t2
sub $t2, $t0, $t1 same as t2 = t0 - t1
addi $t2, $t0, 10 (not binary 2 is decimal 10) so adds ten to the t0 reg value
what if you want to subtract 10 fomr t0? no subi
addi $st2, $t0, -10

Assembly → Machine Code



R-type vs I-type arithmetic

R-Type

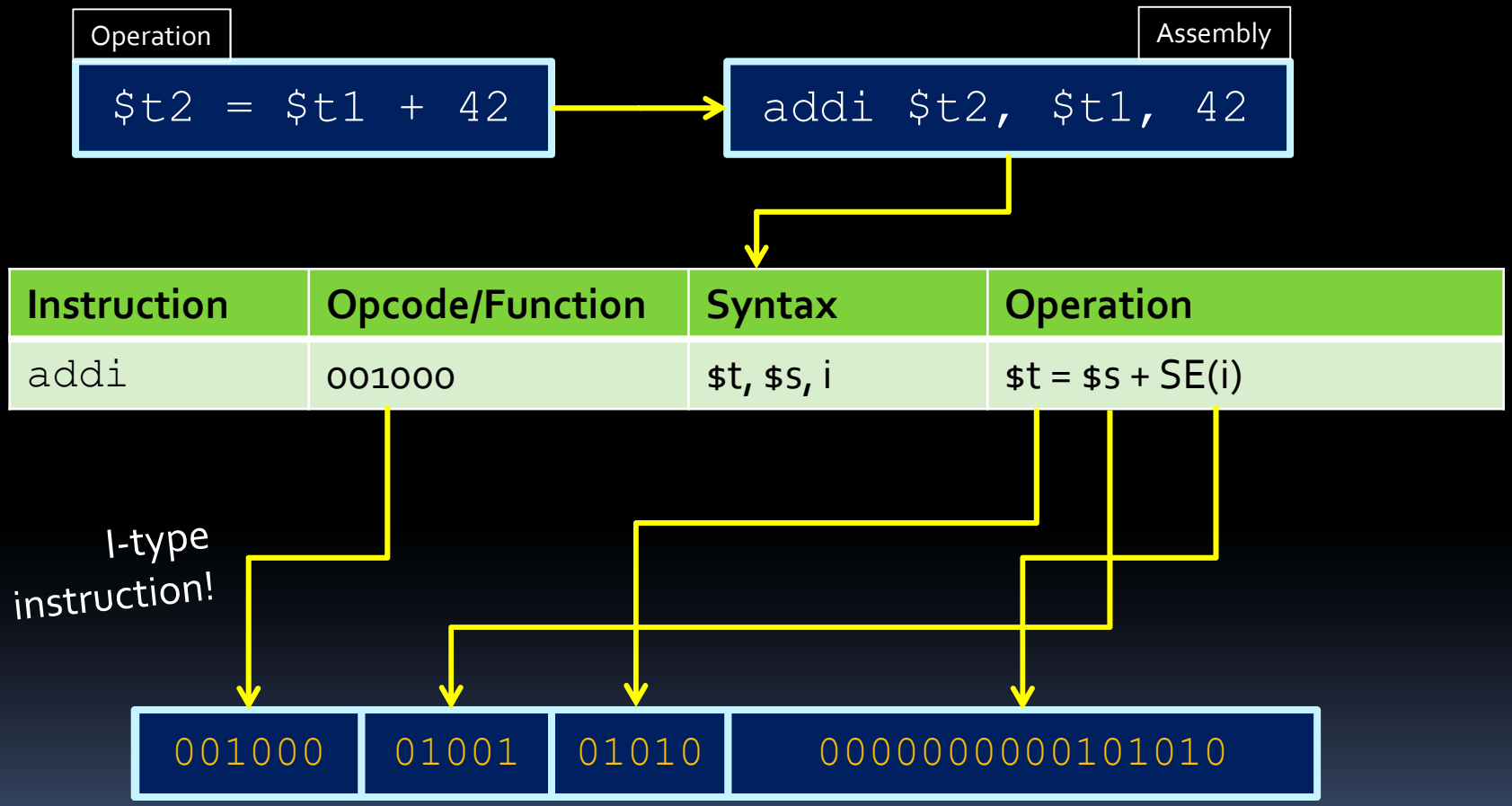
- `add, addu`
- `div, divu`
- `mult, multu`
- `sub, subu`

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).

add valyes - no sign extension fill it in with 0s need to make 16 bit to 32 just need to expand and fill spots with 0s

XoR with -1 to get not
or can use NOR the reg with itself - inverts all the bits

Shift instructions

l or r - left or right shift
a or l -> Arthemtic or logical
are shifting anumber? then need to consider signs (A)
if just a bunch of 0s no matter just shift in 0s to fill in the gaps

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

v = value
in all of these cases where we dont have v then the amount we shift by is just some number that comes from the instruction (SHAMT - shift amount)
thrid reg not used if you use shamt to shift it

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.

m = move
f/t - from and to and hi = hi reg and lo = low reg form the 64 bit reg

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;
}
```

add t0, zero, 10

add t1, zero 1, -. $zero is a special reg cannot be set to anything else
f2 intiaalize is t2, zero, -1

add t1. t1, t2

sub t2 t1 t2

addi t0, t0 - 1

Nov 23 2018

Assembly code example

- Fibonacci sequence in assembly code:

beq- branch - testes here if two things are equal in this 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
```

jumps here and starts the instruction here again

branch statement we will potentially go to another location in mem

branch - may need to go down

if condition met then jumps t J loop step tests the condition first otherwise goes to next line that add

j -> jump , when it gets to this line jump to the statment named loop

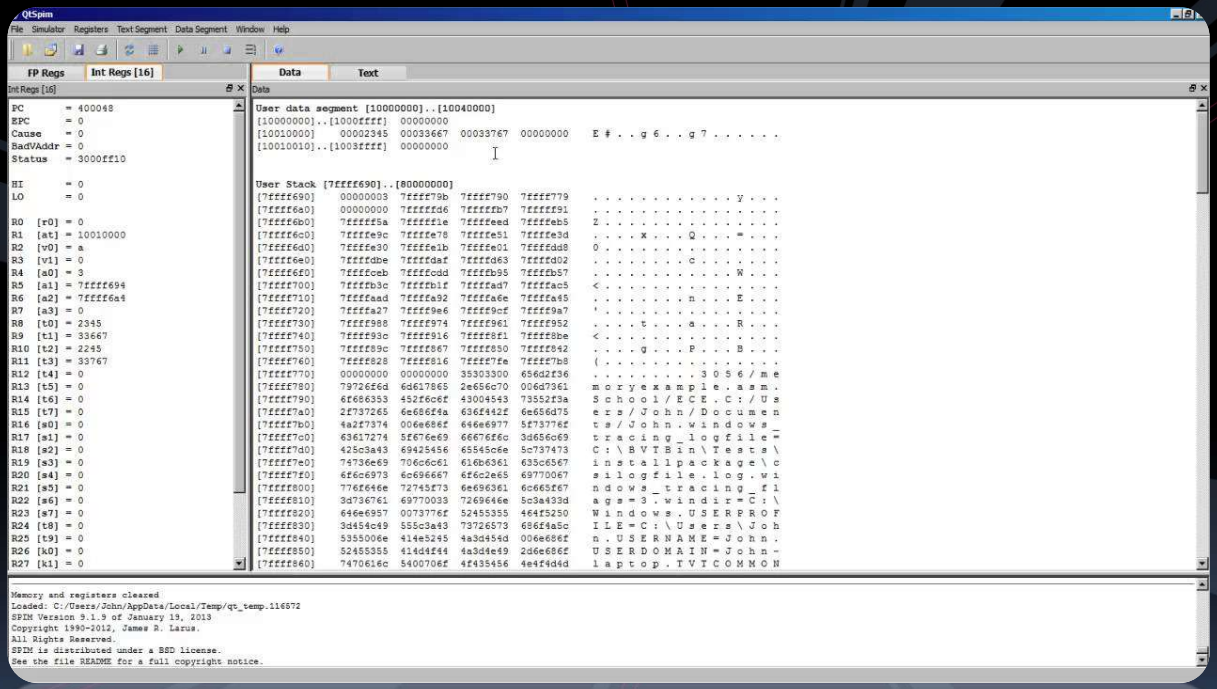
Making an assembly program

- 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

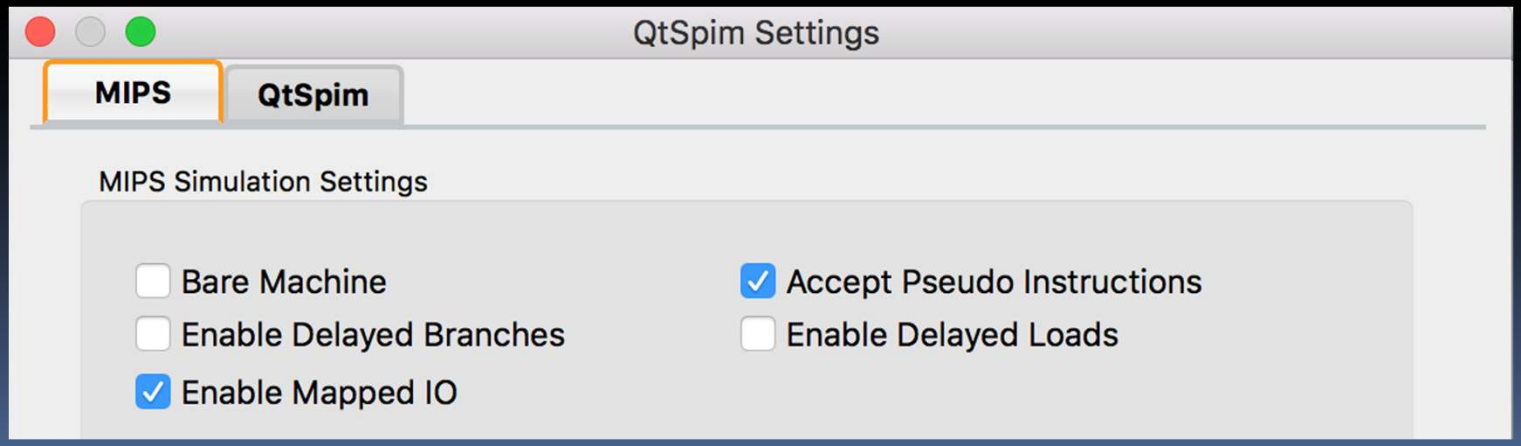
aka: QtSpim

or MARS???



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.



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.

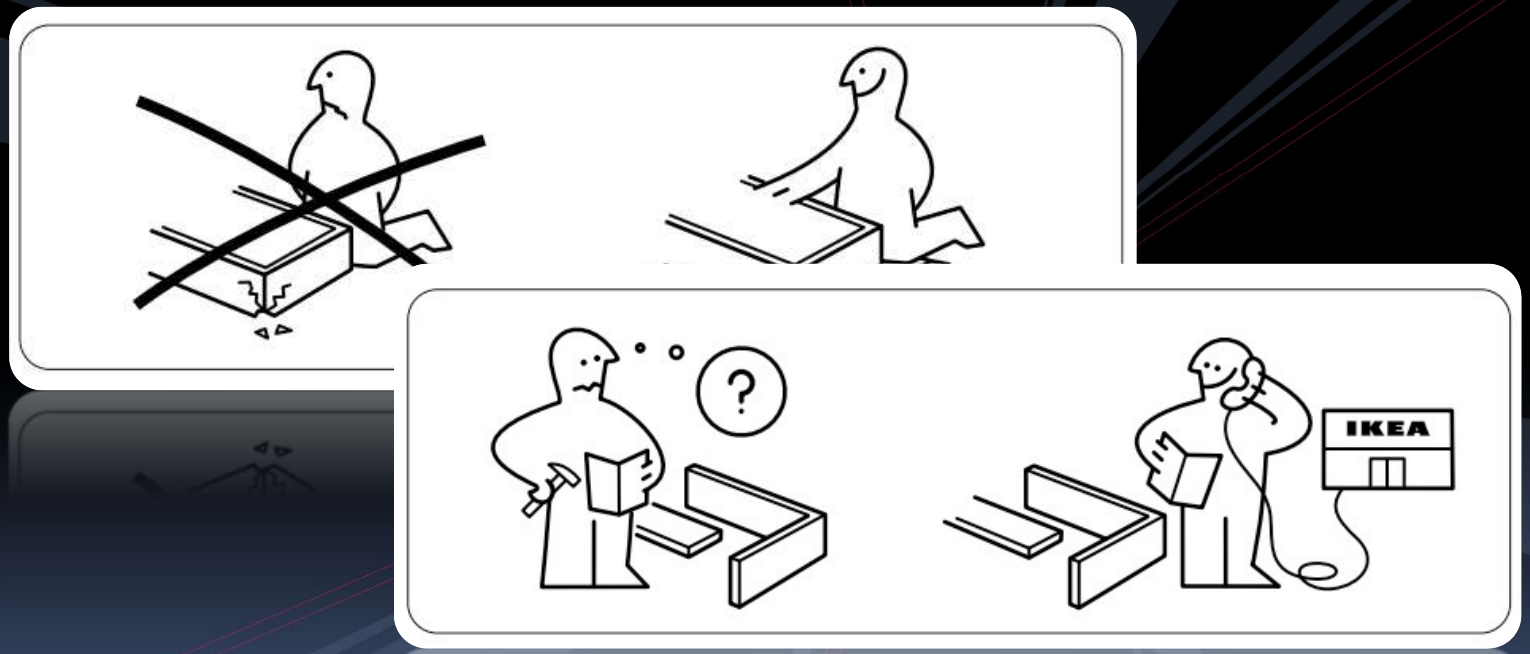
QtSpim – Quick How To

- 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

checks if two thinsg equal

if the two reg are equal go t the label does this by incrementing the PC to go to the label

branch i

if greater than 0

branch of less than 0

branch if not equal

if not equal jump to tho this label

- 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
        ...   case when not equal up here
        ...                    # if $t0 != $t1, then
                               # execute these lines

 end:   ...   labels have to be uniuqe across entire program
        ...   so only put labels on lines of sig that need to be jumped to
                               # 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
```

case whe equal here

this is case when not equal

- 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 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).

branches diff than how jump stores tings - jump replcecs the program counter info
here it doesn't use it just adds it to the immediate and gets a new location
biggest diff - jumps have address and puts straight into counter
branches - say how far from PC do you need to go i.e. via immediate

diff arcteutcure do diff thinsg i.e pc +4 yjhen opp then IR
others do op first then incremenet then IR

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.

for this course take do this (*****, we have a shift by 2 because the 4 byte thing branches updated PC -> same thing happens here - no store number of bytes sure assume # of instructions need to move - so we store everything but the last two bits and when we updated the P we will add the 2 0s
shift by 2 in digaram - is only used when we are dealing with branches (the big mciro processor digram)

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?

if cond not statstifed

what is range how far thing can jump (EXAM)
2^26 is how far a jump can go but branches dont have 26 branches available
but a branch has an offset taht is 16 bits -> 2^15 instructison forward and 2^15 back so 2^16 in total

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

jump blindly

jump and link

not jumping as part of a loop - jump into another function or a set routine

rtake PC and set to first line in function , when the function is done it wil jump back to the orignal, if normal jump no way to figure out how to jump back to roginal fn - but here use one of the regs to store original PC and then jump into fn so when fn is done go to reg and restore control back to original prorgam i.e 31, juses JR to jump to reg to go back. what if you have deeper prorgam - i.e function has to call another function (LATER) the answer to this is stack

jr = jump register

so this sets the PC to be the address stored at the target reg

so jumps to location stored in a register

what is bad address stored in reg? not an divisible by 4 number - would throw and invalid address error (exception - blue screen of death)

rtype will be given table in exams says what type there are

- `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
```

if no jump the condition when both not qual willrun both since will run all lines one afte rthe other

if equal go down here

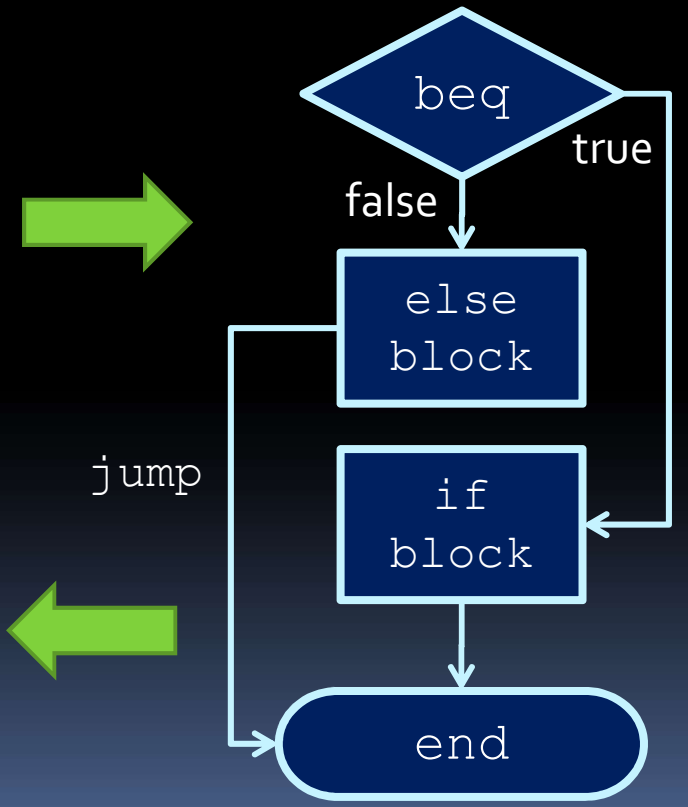
- Or branch on the else condition first:

```
#  $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;
```



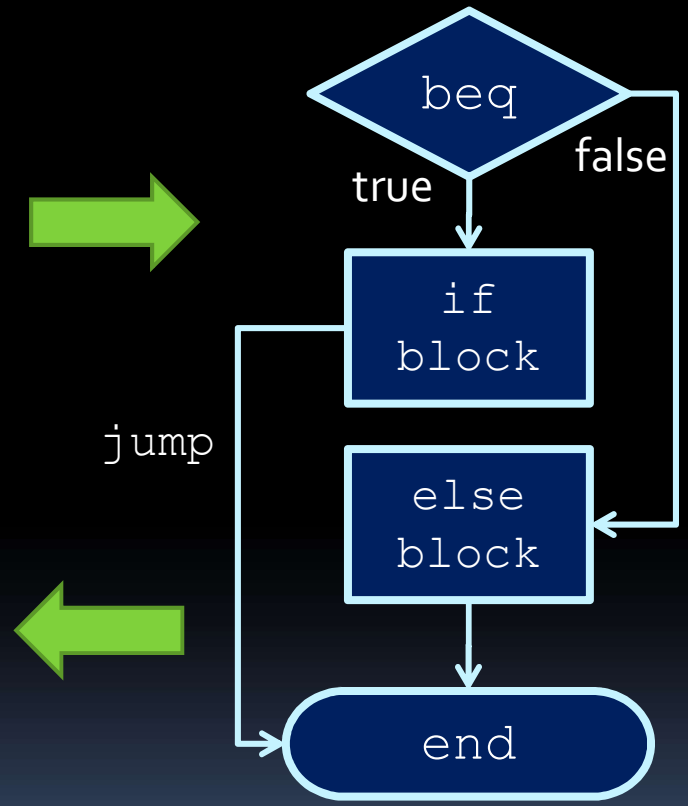
jump

```
#  $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?

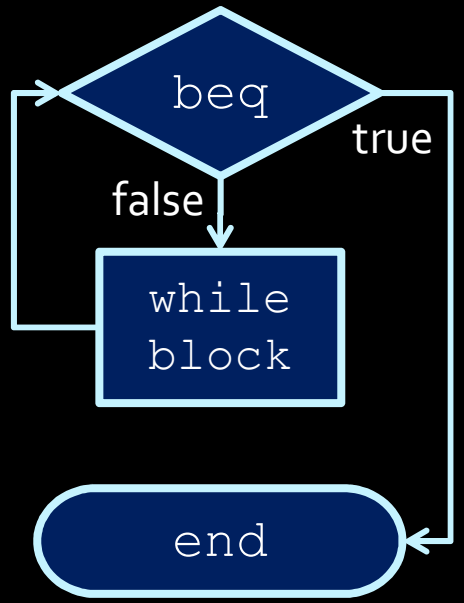
```
if ( i == j && i == k )
    i++ ;  // if-body
else
    j-- ;  // else-body
j = i + k ;
```

both of them have to tbe true

```
#  $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++;
}
```

in assssembly cannot tell diff between teh for and the while loop they are the same thing at this level

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:

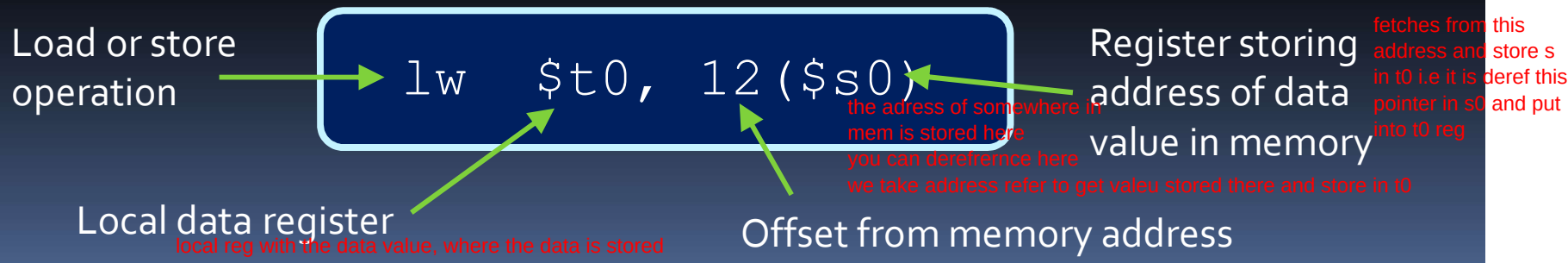
```
#  $t0 = i, $t1 = j
main:    add $t0, $zero, $zero        # set $t0 to 0
         add $t1, $zero, $zero        # set $t1 to 0
         addi $t9, $zero, 100         # set $t9 to 100
START:   beq $t0, $t9, EXIT           # branch if i==100
         add $t1, $t1, $t0            # j = j + i
UPDATE: 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



Loads vs. Stores

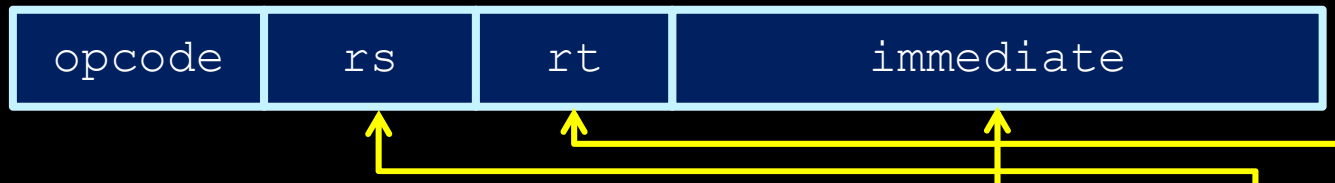
- The terms "load" and "store" are seen from the perspective of the processor, looking at memory.

loads you read from mem and load into reg

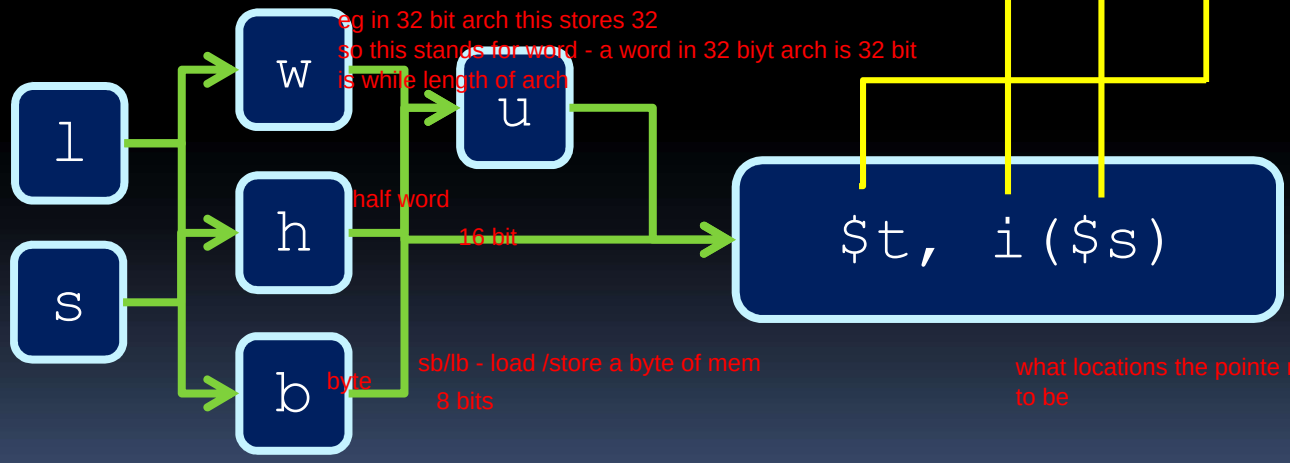
- Loads are read operations.
  - We load (i.e., read) from memory.
  - We load a value **from** a memory address into a register.
- Stores are write operations.
  - We store (i.e., write) a data value from a register **to** a memory address.
  - Store instructions do not have a destination register,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



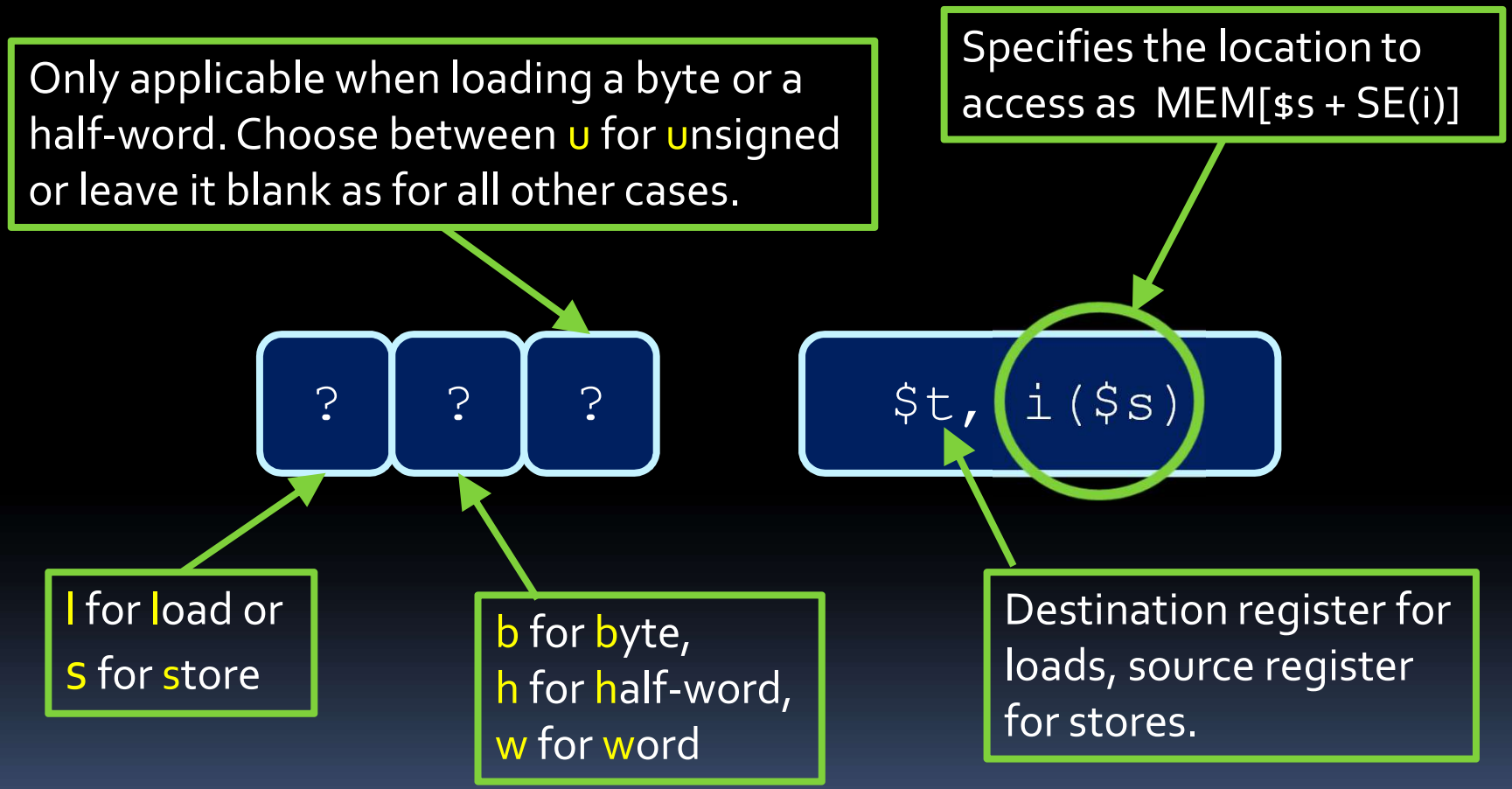
Load & store instructions

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

u what is the u???
the u is unsigned
why do only some have unsigned and other no have?
when load half word - and you have cap to store full word
you need to fill the
rest of the space
if full reg load half a word from mem - signed extension
if neg number
a word will fil whole thing automaically so no need to extend
when it comes to storing no need u here because
nothing needs to be extended only take up the space you
need, go into as much space as you need in mem

- "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.

all words and half words are half aligned allways at address divisble by 4 (word) or 2 (half word)

instructions alway at addresd divisible by 4

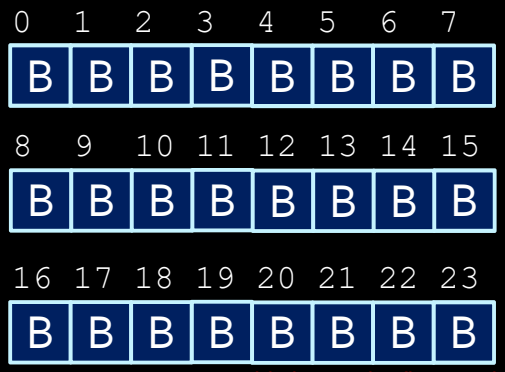
in mem can grab a byte/half word or a word
memeory is always aligned you can can only grab half words from even locations
cannot grab 16 butes from odd number address

if we want tos store number 258 where woulld they sit, should store them in258 order i.e 2 in first then 5 then 8 but does it this work? will it be backwards? should siwtch it so store 8 in lowest sig bit i.e 852? what is the right answer?
two ways to store things in mem
which way is clear winnner?

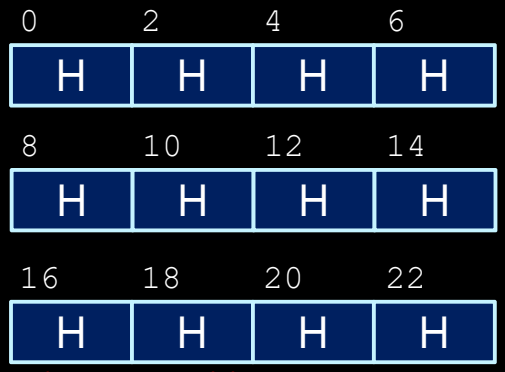
Memory alignment

only can get words from adresses divisble by 4

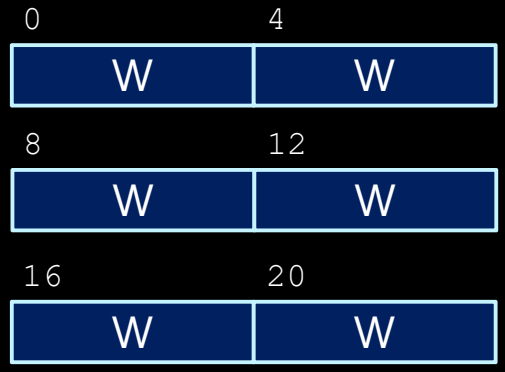
Byte alignment



Half-word alignment



Word alignment



this is ony the first 24 since not enough to store eveyrhting

- 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?

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

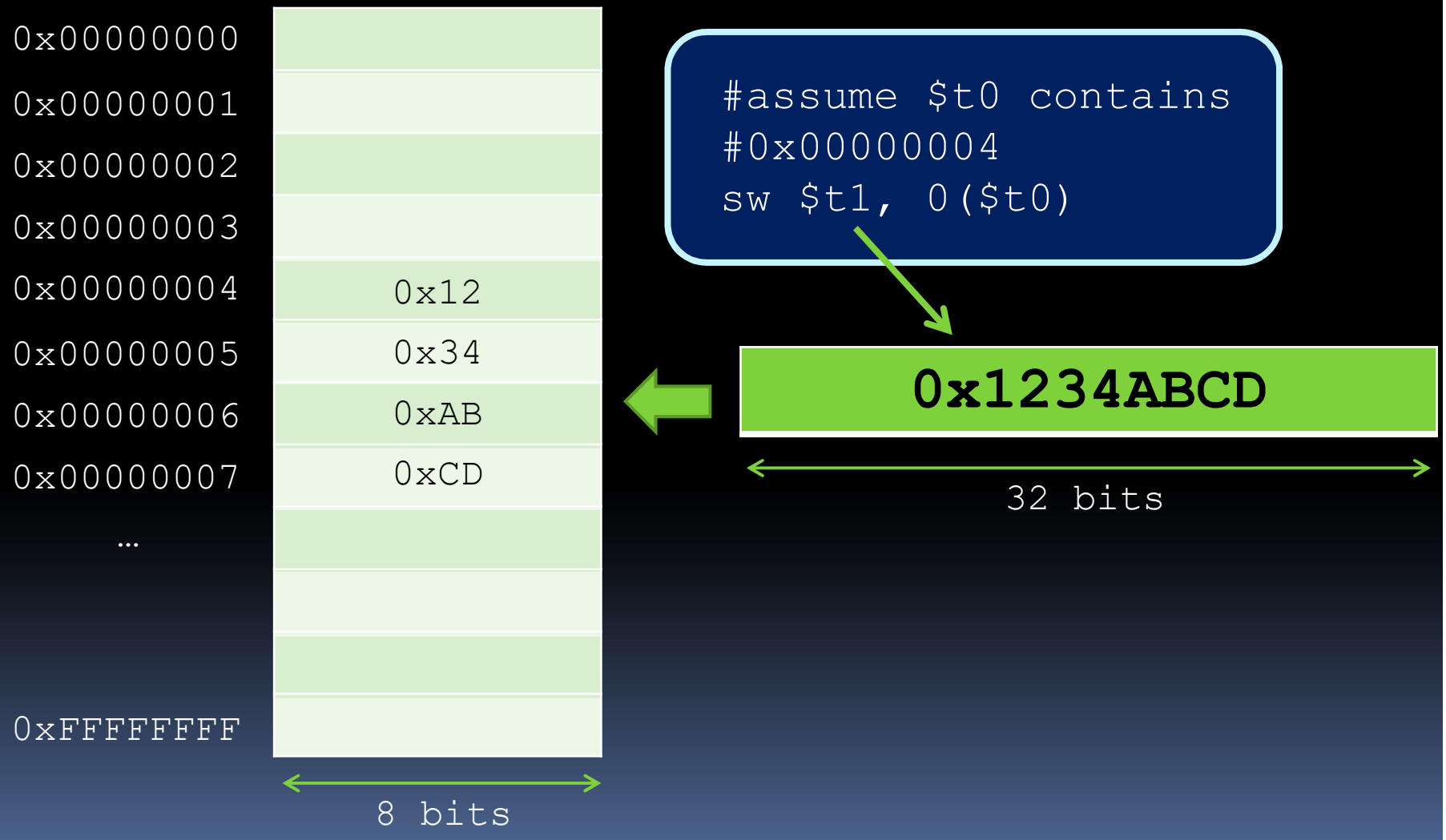
Little Endian:

Byte D	Byte C	Byte B	Byte A

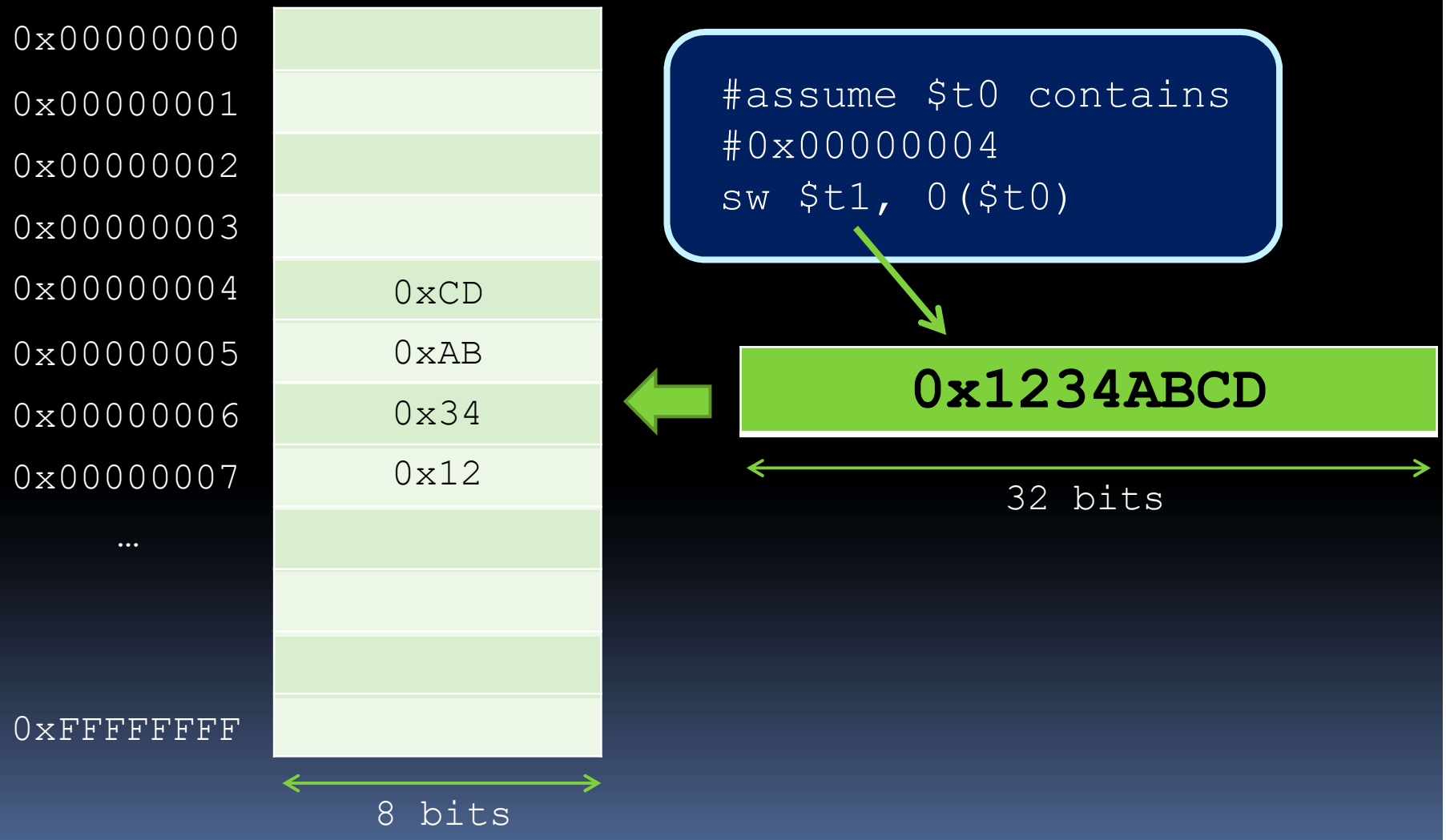
Big Endian vs. Little Endian

- 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



Little Endian Example



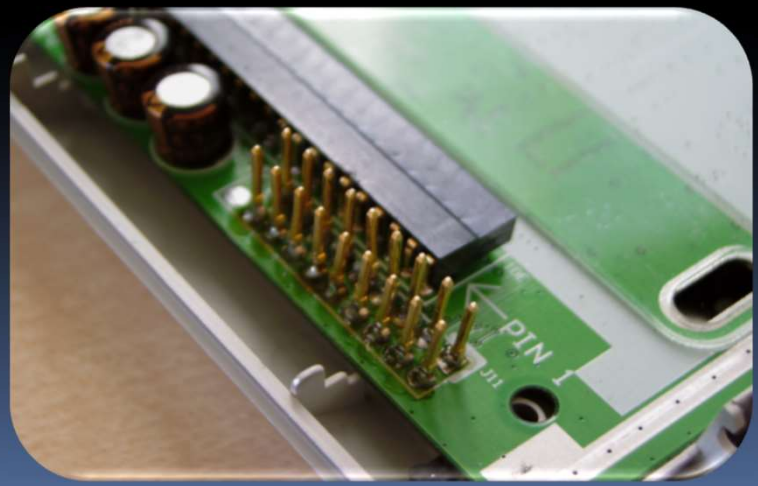
MIPS Endianness

no standard on which one to use

- 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.



just know this table exixts and this is how it works no need to remember this table though

It's a trap!

Instruction	Function	Syntax
`trap`	011010	i

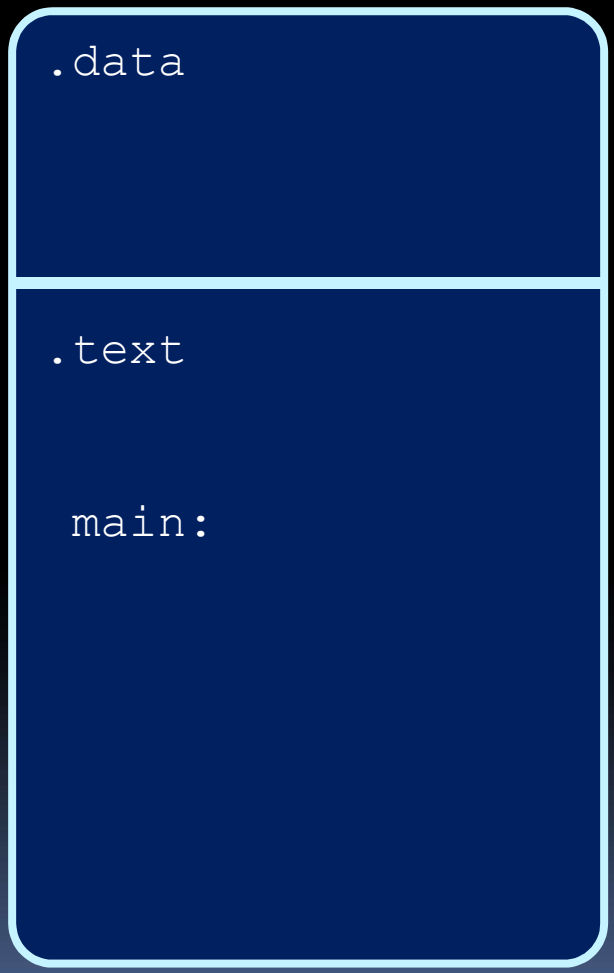
- Trap instructions send system calls to the operating system
  - e.g. interacting with the user, and exiting the program.
- Similar but not quite the same as the `syscall` command.

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**

when write a program it is structured this way
data at the top
and then text section with all the code and main will be the starting point like in C

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)`

need label to say how big need how big it will be

```
# 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'
```

neeed quotes otherwise might not know what it will be
there is ascii lok up table that will tranlaslate characters into bytes

```
# 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
```

like c malloc - memoery alloocation

will go through look for 40 bytes and put a table here and will have a memoery manager to make sure nothing else is declared in this space (OS)

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.

at assembler time use this reg to use psuedo 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

make instruction called la - which loads a reg with a partticular value

- `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**

store in upper bits

    - **"immediate2" represents the lower 16 bits of the memory address label corresponds to.**

pesudo instrcution allow syou create new instruction that will expand into whatever lines needed to make it work

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

in assembly can have arrays but no special array

Arrays!

create some space to give hoever many bytes you need

```
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.
```

declarations at the top

give space for 100 ints because 4 bytes for int

char is 1 byte
a int is 4 byte
i.e 32 bits long

text section

data values all stuck in mem need to do something first i.e get them from memory first - if stuck in mem have to fetch them first - first step get things from mem so load address from A and B so that you have what is like a pointer to start of mem ->so that you can fetch values have two arrays get value at start of B and add offset to it and detch value (so in this case add multiple of 4)

increment t0 until you get to 100

use t0 to inidcate how far to is down shift by 2 to get posiiton

get addres soffset

number of regs: need 1 reg to store locaiton of A onr for location of B, one to store where in A are you (what index) one more index in B and one for offset which will come form I which is another reg and another reg to tell you what value i will hit until it stops - need 7 regs

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)
```

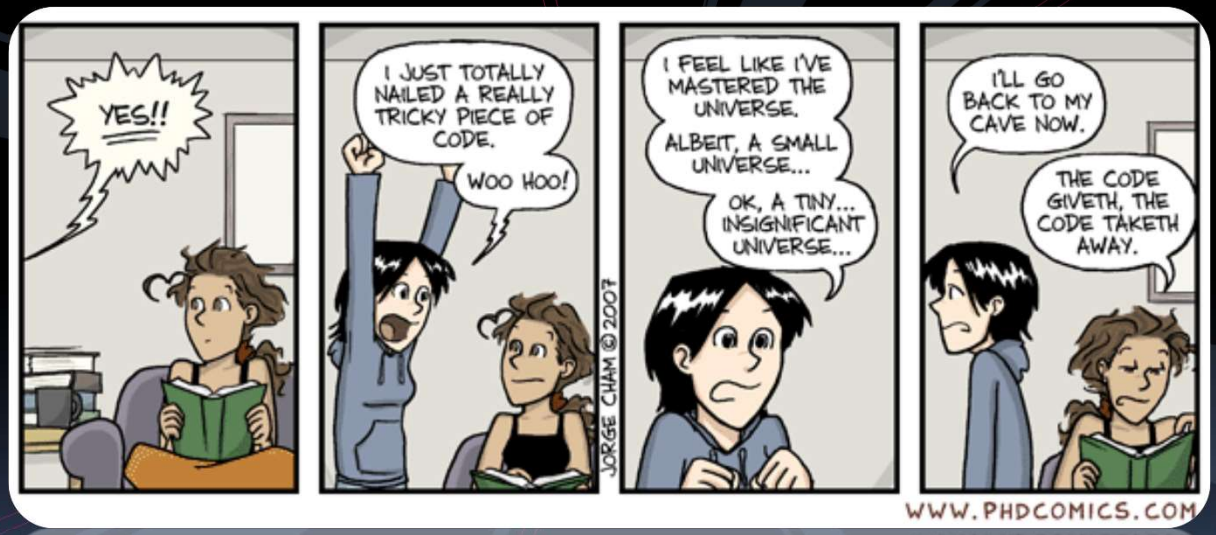
we store five things in t1 then into memory
t1 is like a staging ground before it is dumped into memory

this number 8 is an offset
go eight spots forward from that location
these are used a lot n structs
so you know where things are stored eg steves name is at 0 and then 4 away you have his birthday 8 away you have his address or something

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



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

almost all rdecalrations done at top and then what ref we are using is defined at the start - need to allocated and intialized and document this first

```
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

                .text is missing here

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
```

when this things ic omplicinggoes to first twp lines and assigns things to i and j wherever it is avai;able, anywhere that envounters i and k it subs the address for i and j

anytime see these addresses use the address for i and j, so these addresses are stored in t0 and t1
you have to load the address of bascically any variable you will use

interally for the processor

externally for anyone using it

debugger will see these twp lines but not the two lines after it? so these one intialize ones in mem and next two intialize the reg

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

special regs a0, a 1, a2, a3 are input regs and output ones v0, v1
four of them for a and 2 output ones

- 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?

whenever you have a function it will take a cetrian amount of room on the stack
what happens if too many functions? and goes into data sections? - in a recursive fn that has no base case will keep pushing things onto teh stack which get a stack overflow goes firther and further into memeory and eventually hits data memeory - and now it's obviously did something bad - get rid of the function now taht have the label of fn that casues teh overflow

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.

we are copyting thinsg over until we get to the null terminator

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;
}
```

we need to copy values from y into x what will we need??? what do we need to store?

we need a pointer to the two arrays (the front of both arrays)you need two indcies for thes two
need a pointer to tell when to stop - need one set of pointers fro front and ne that is mobile until hit null (index)
you need to store the value - have to physically store him and then dump into the nother array take the reg that stores the number into the other array (intialization)

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:

stack pointer sits on top of stack right now and then move to get value x and then address of y

```
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:

sp move over a bit so I can dump into 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
         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
         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
         sw     $t0, 0($sp)      #  the stack
         jr     $ra              # return
```

initialization

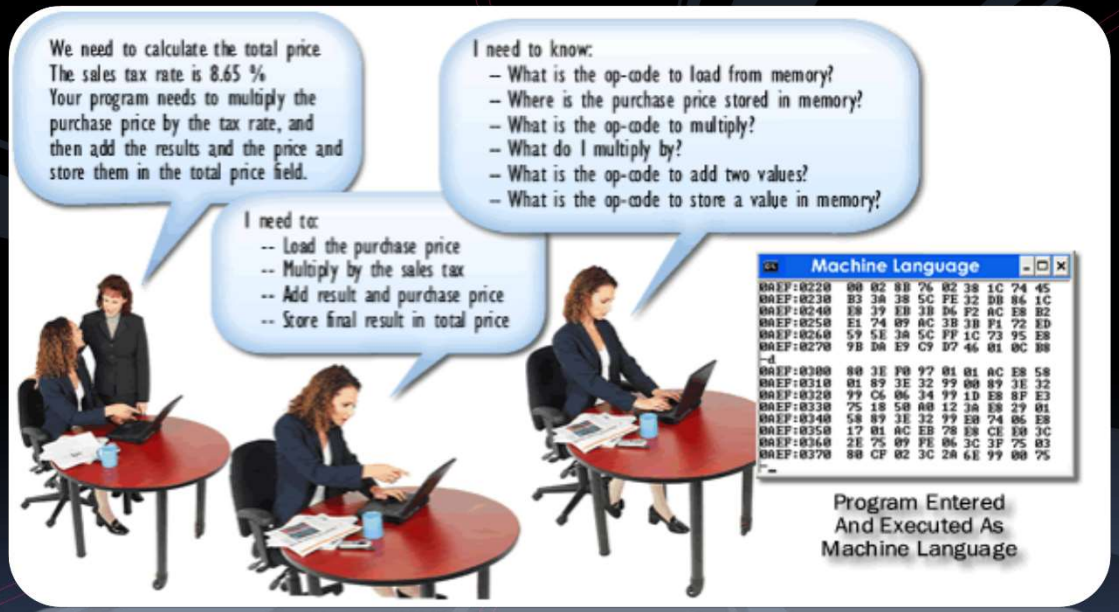
main algorithm

end

load into A0 and a1 the addresses (looking for them in the stack)

check for end condition

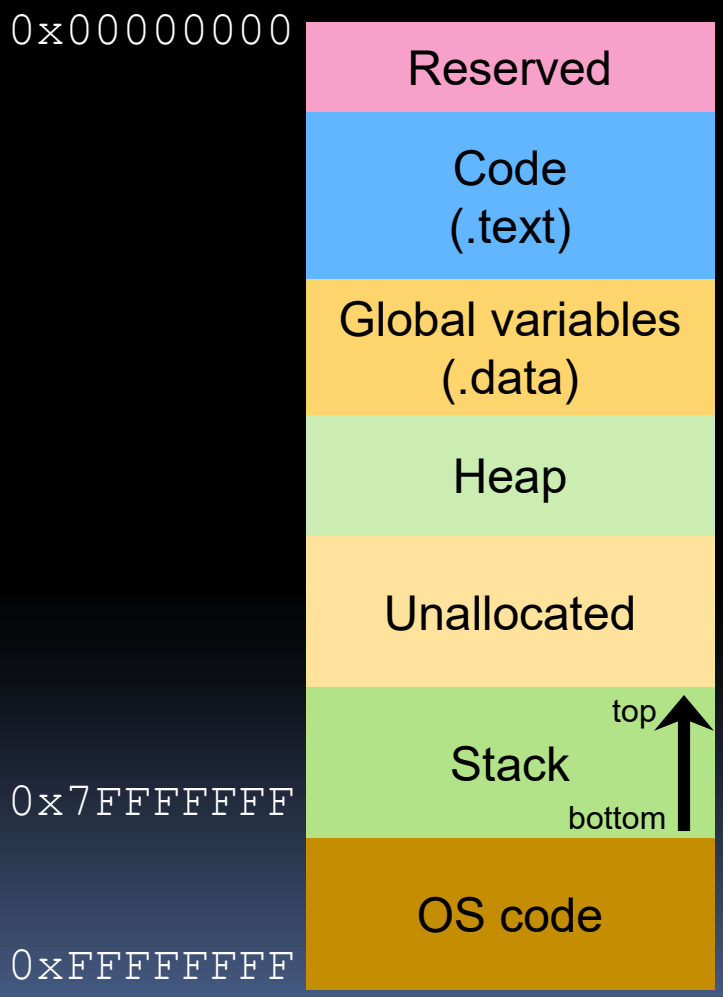
Function Considerations



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

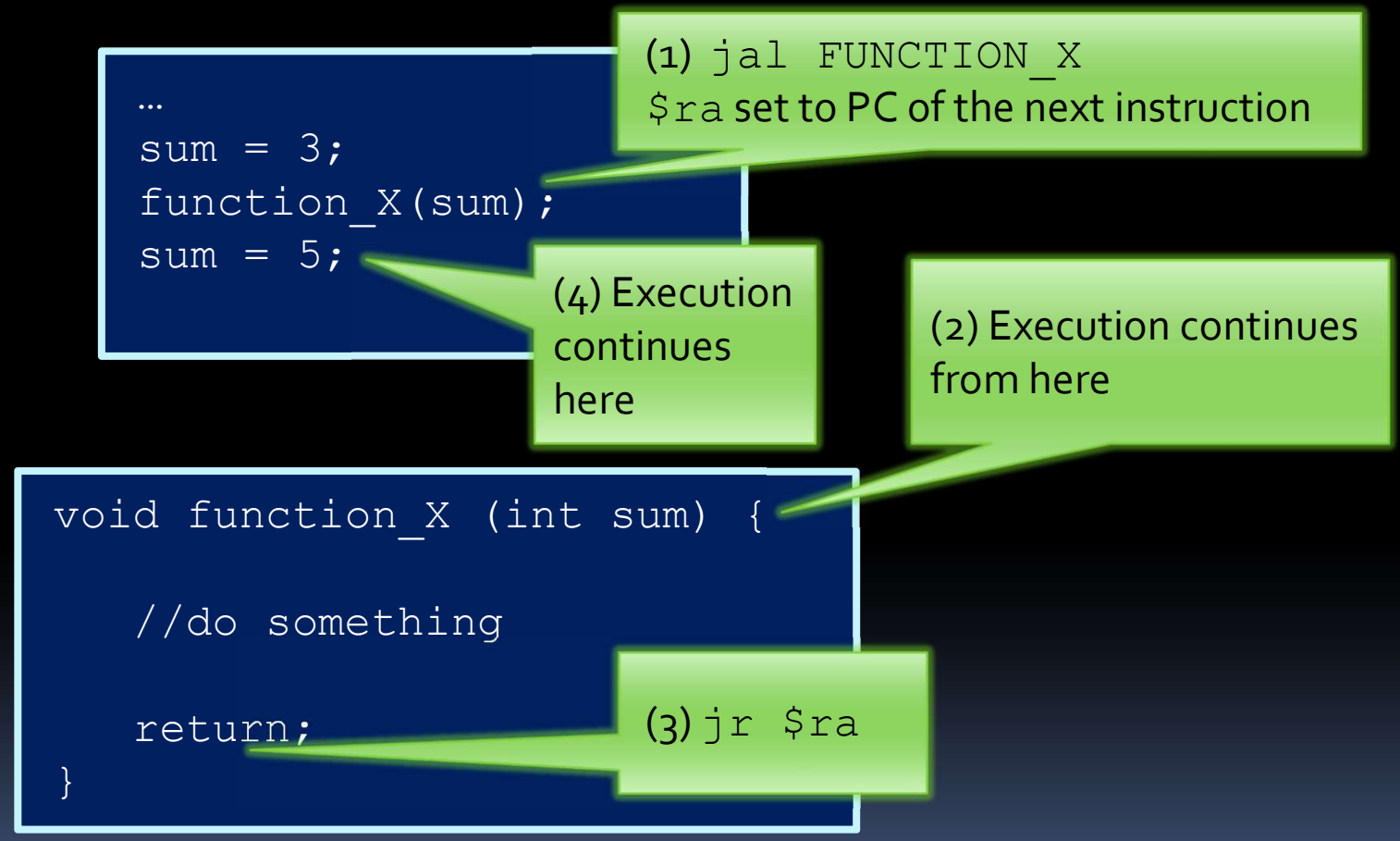
```
…
sum = 3;
function_X(sum);
sum = 5;
```

```
void function_X (int sum) {

    //do something

    return;
}
```

Function Calls – Cont'd



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)

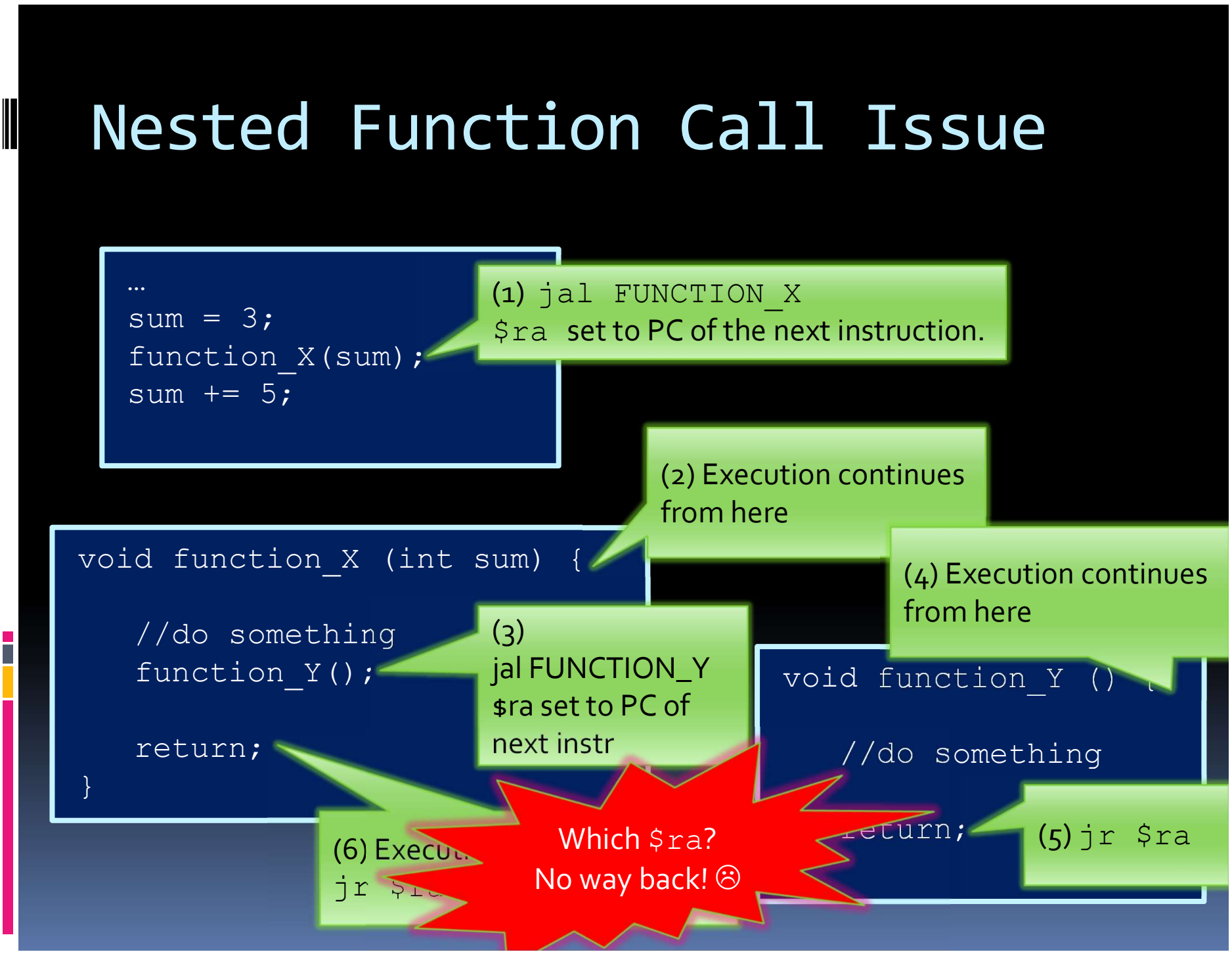
```
.data
RESULT: .word 0

.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



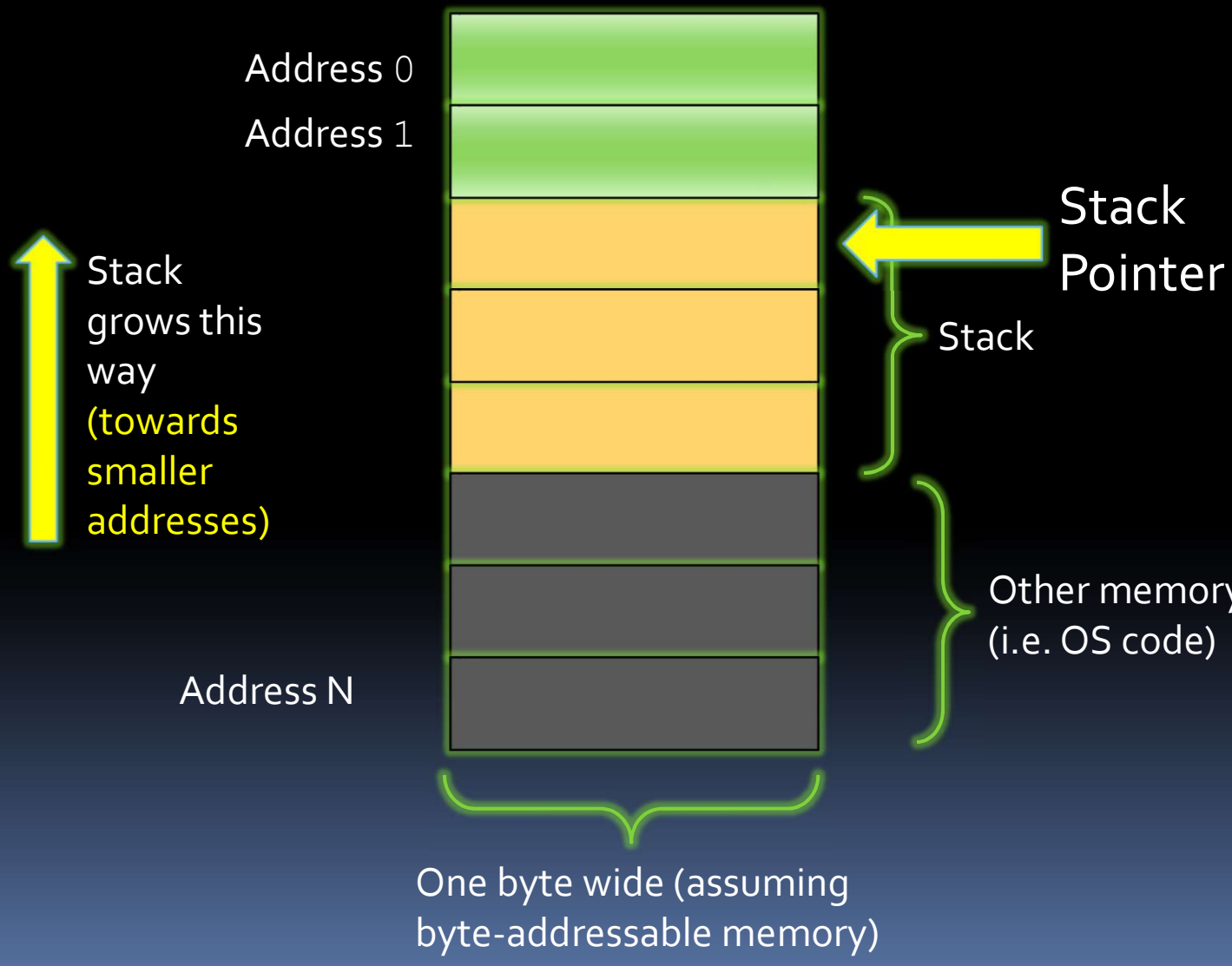
The stack to the rescue!

- 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).

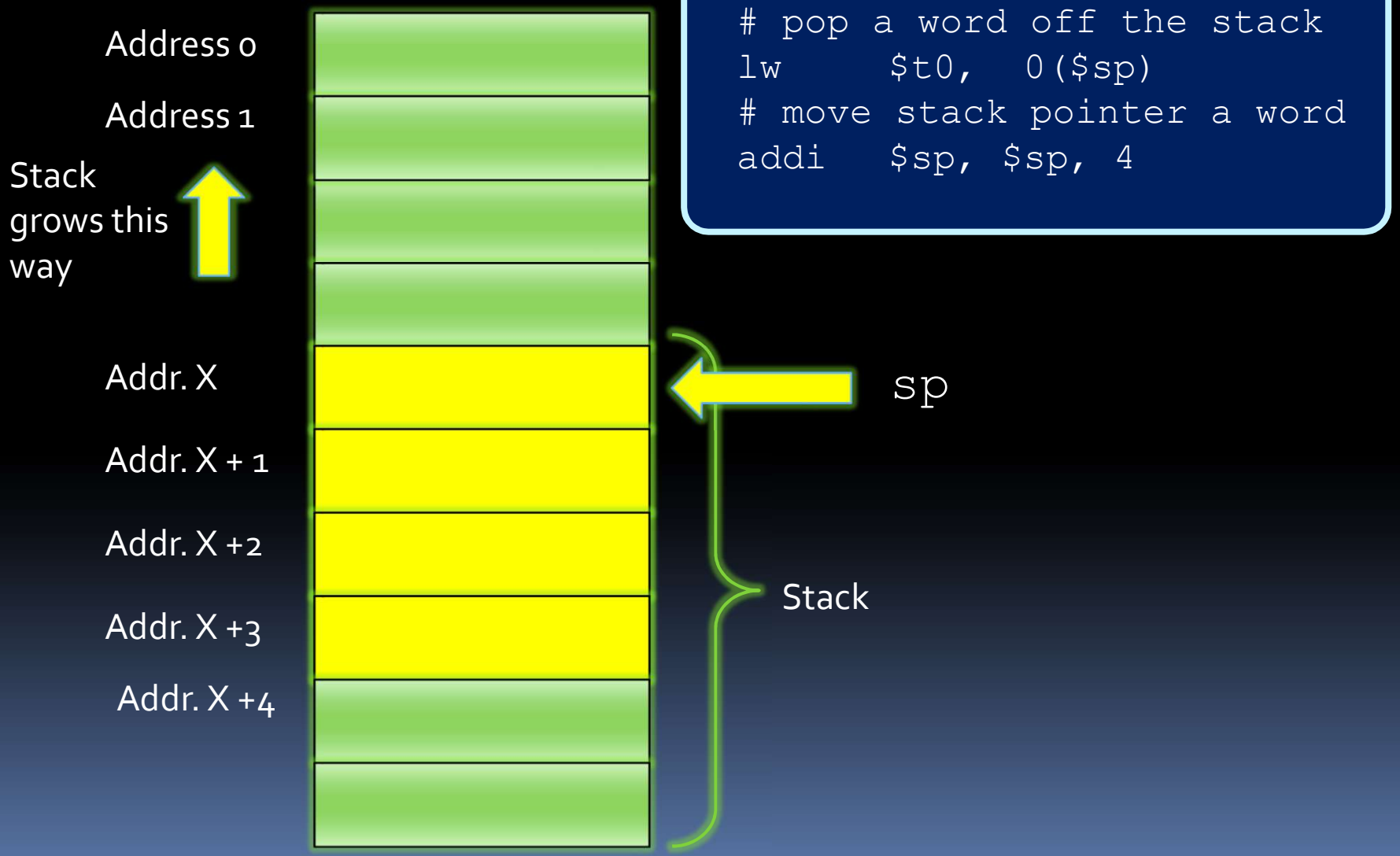
* *As mentioned before there are some predefined registers used for the function arguments and return values; the stack is used if this number is exceeded.*

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

The Stack, illustrated

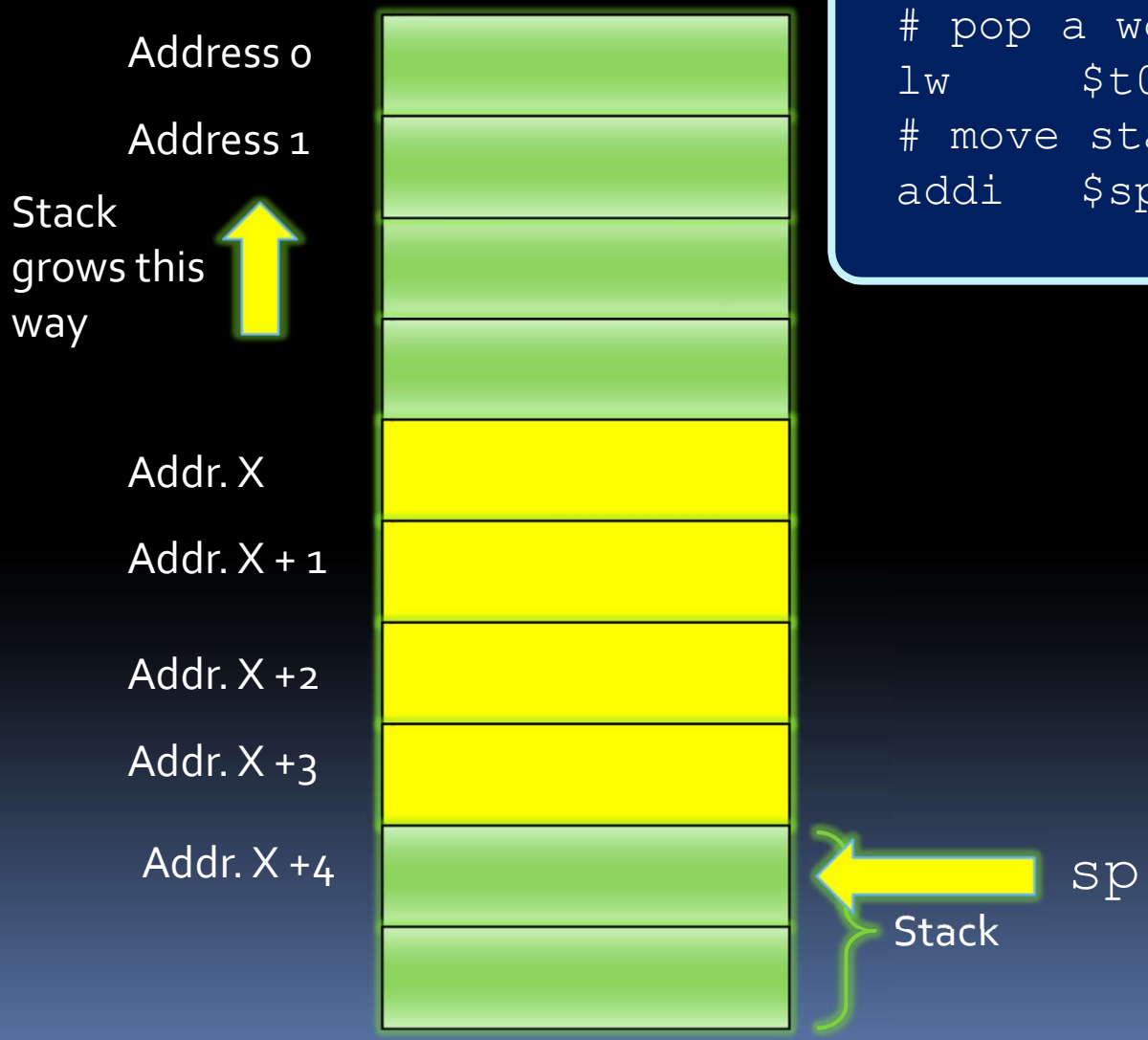


Popping Values off the stack - Before



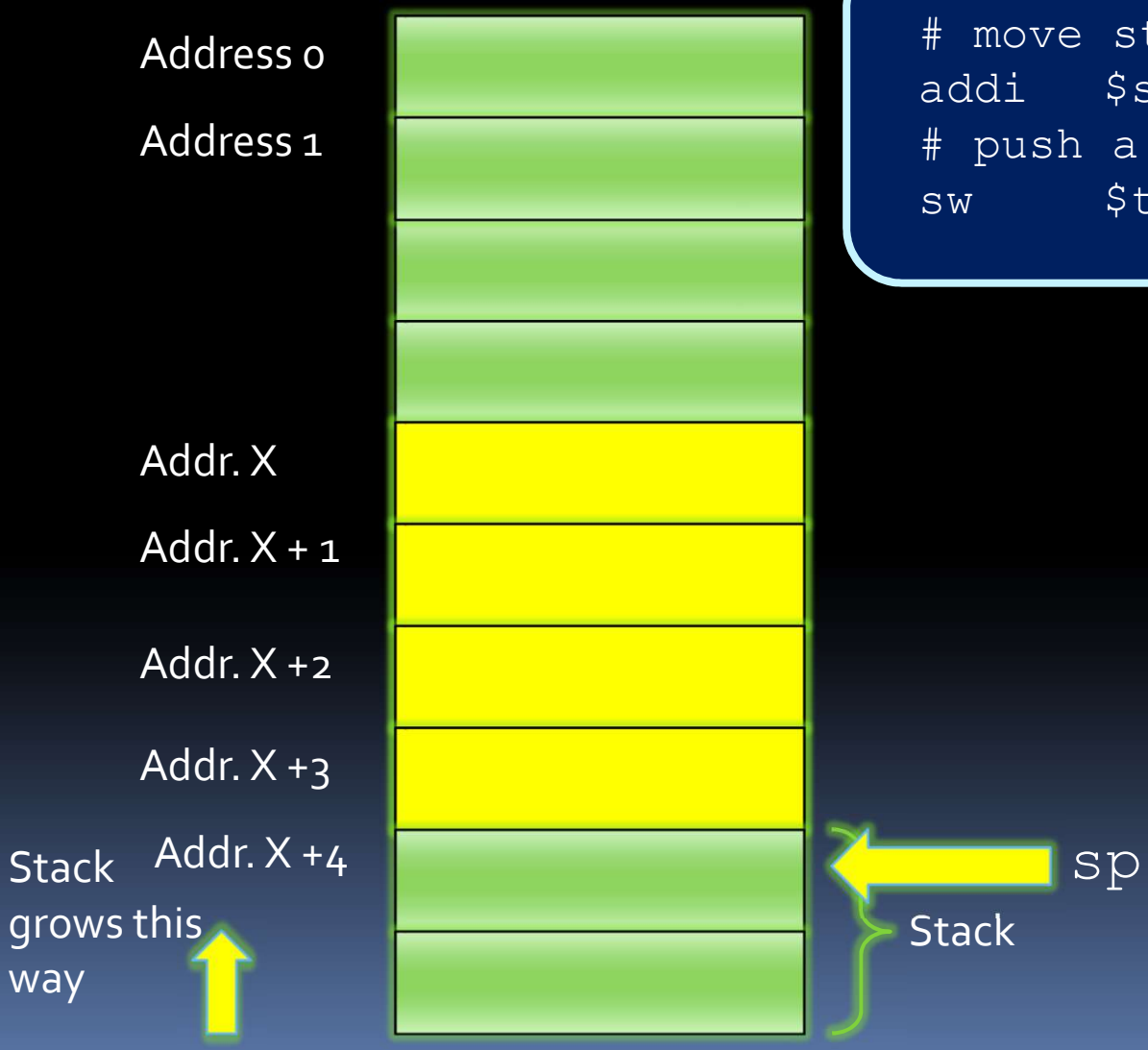

```
# pop a word off the stack
lw     $t0,  0($sp)
# move stack pointer a word
addi   $sp, $sp, 4
```

Popping Values off the stack - After



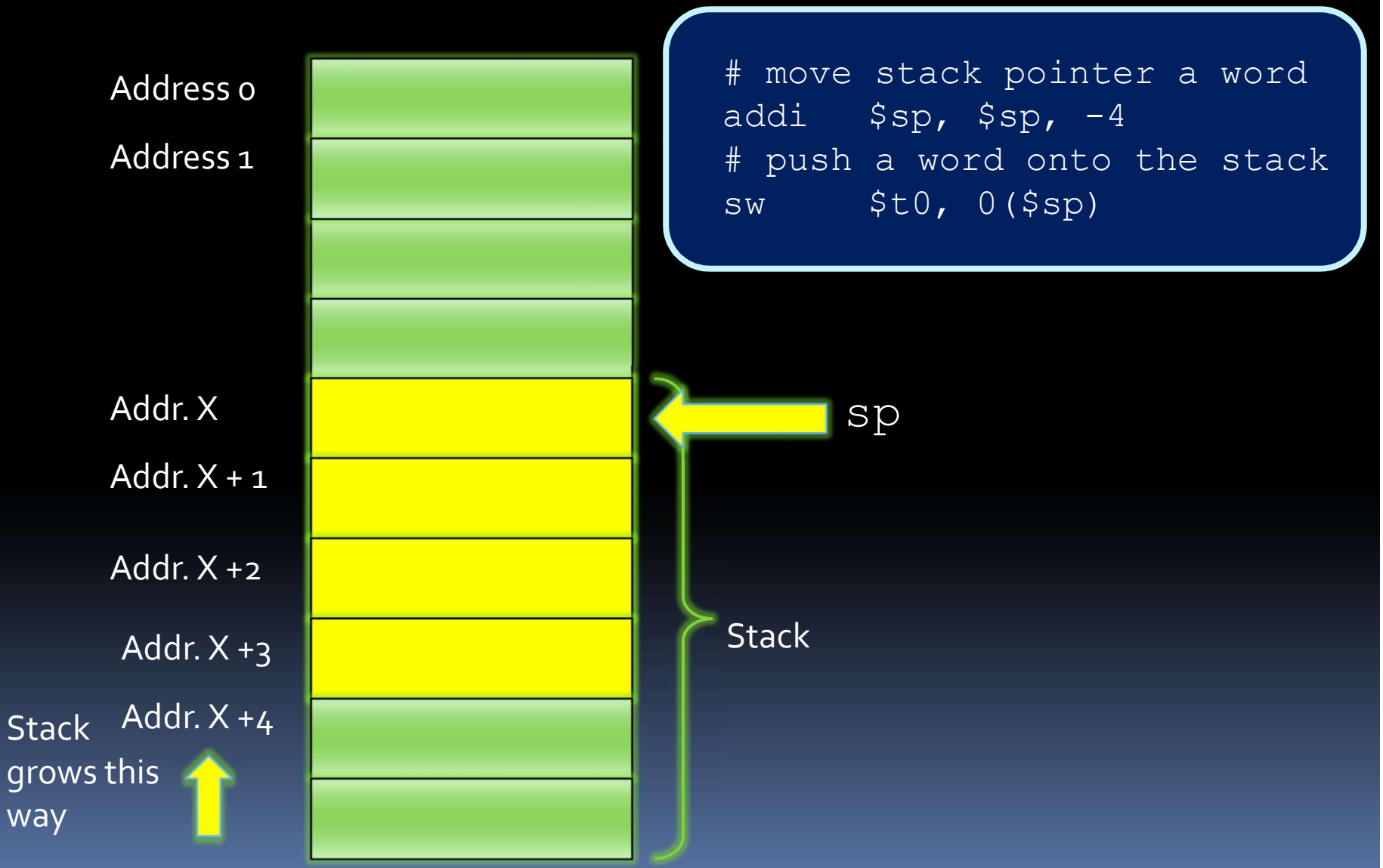

```
# pop a word off the stack
lw     $t0,  0($sp)
# move stack pointer a word
addi   $sp, $sp, 4
```

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)
```

Pushing Values to the stack - After



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.

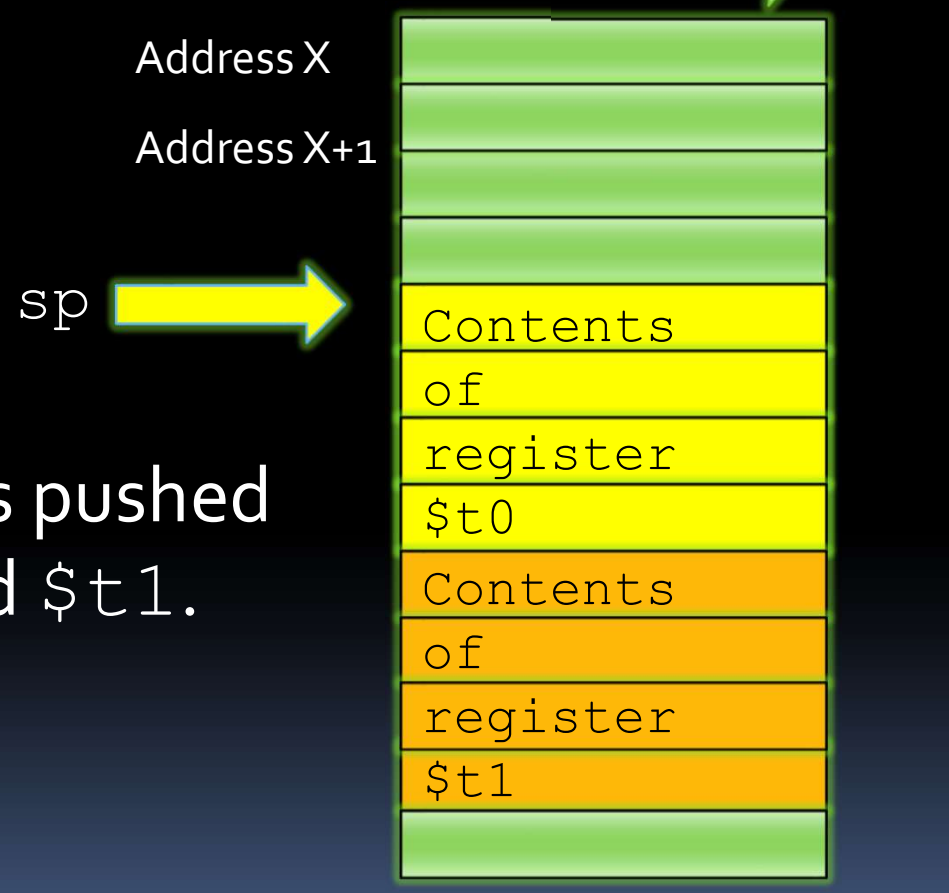
Stack storage example

- 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 $to-$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 $so-$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 $to-$t9?
    - No action needed. It it the responsibility of the caller to ensure there registers are not modified.
  - Using $so-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

```
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

- 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

n → $t0
n-1 → $t1
fact(n-1) → $t2

factorial(int n)

```
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
```

n → $to

n-1 → $t1

fact(n-1) → $t2

factorial(int n)

```
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
```

n → $to

n-1 → $t1

fact(n-1) → $t2

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)

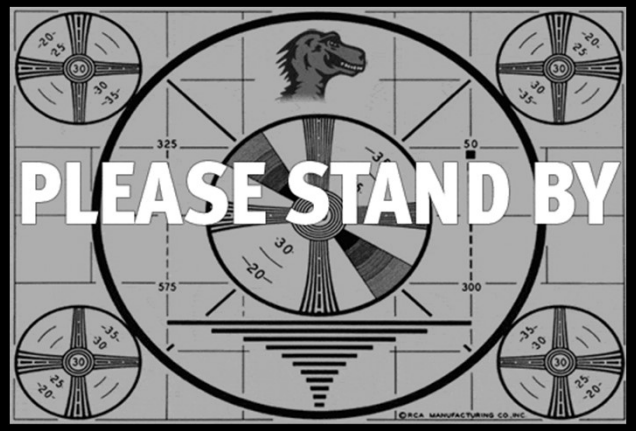
```
(continued from part 1 - returning from recursive call)
            lw     $t2, 0($sp)           # get return value
            addi   $sp, $sp, 4           #   from stack
            lw     $ra, 0($sp)           # restore return
            addi   $sp, $sp, 4           #   address value
       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

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