# Computer Architecture (Wk 1-5)

# Week 1 - Number Systems

Indicate the number system by putting a subscript at the end (e.g. $1101_2$ for binary - base 2)
A new digit is introduced when greater than the last digit of the base system.

Numbers are used to represent both instructions and programs stored in memory where they're read/written.

## Decimal System

- 10 digits (0123456789)
- Example: 193 = $1 * 10^2 + 9 * 10^1 + 3 * 10^0$
- Based on position
- You add a 0 to the right when multiplying by 10
- Extendable to represent fractions / rational numbers
- Used in everyday life

## Binary System

- 2 digits (0 and 1), where a binary digit is known as a **bit**
  - 0 = OFF state
  - 1 = ON state
- Example: $1101_2 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 13_{10}$
- You add a 0 to right when multiplying by $10_2$ ($2_{10}$) - doubling a number
  - e.g. $1101_2 * 10_2 = 11010_2$*
- denoted by prefix $0b$
- Used in computers

## Octal System (not covered)

- Base 8 system (01234567)
- denoted by 0*

## Hexadecimal System

- Base 16 (0123456789ABCDEF)

    - A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
    - A = 1010, B = 1011, C = 1100, D = 1101, E = 1110, F = 1111
- denoted by prefix `0x`
- Easy to convert to binary and vice versa
    - $101000011100_2 \rightarrow 1010\ 0001\ 1100$
    - 1010 = A
    - 0001 = 1
    - 1100 = C
    - therefore result is 0xA1C (equiv to 0b101000011100)

# Binary Arithmetic

Same as with decimal arithmetic

$$\begin{array}{r} {\scriptstyle 11} \\ 1101_2 = 13 \\ + \\ 0110_2 = 6 \\ \hline 10011_2 = 19 \end{array} \qquad \begin{array}{r} 1101_2 = 13 \\ \times \\ 0110_2 = 6 \\ \hline 0 \\ 11010 \\ 110100 \\ 000 \\ \hline 1001110_2 = 78 \end{array} \qquad \ldots$$

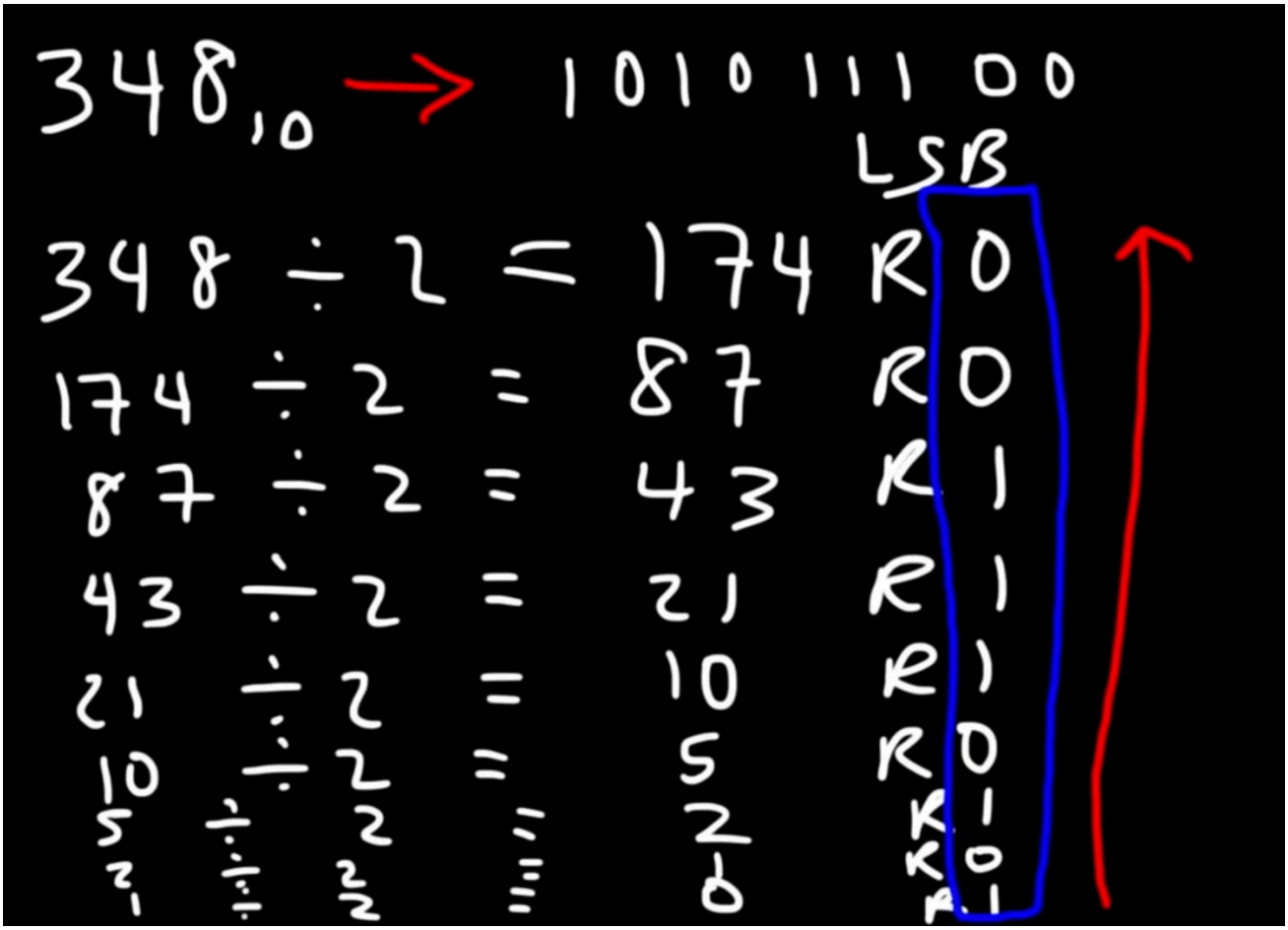
# Converting between Number Systems

**General step**
Decimal -> Binary, Decimal -> Octal, Decimal -> Hexadecimal

- successive division
- Repeat until less than the base value
- obtain result by working upwards (MSB [leftmost] -> LSB [rightmost])
  Vice versa
- multiply each value by the base and power (distance from end)
- total the answer to get binary

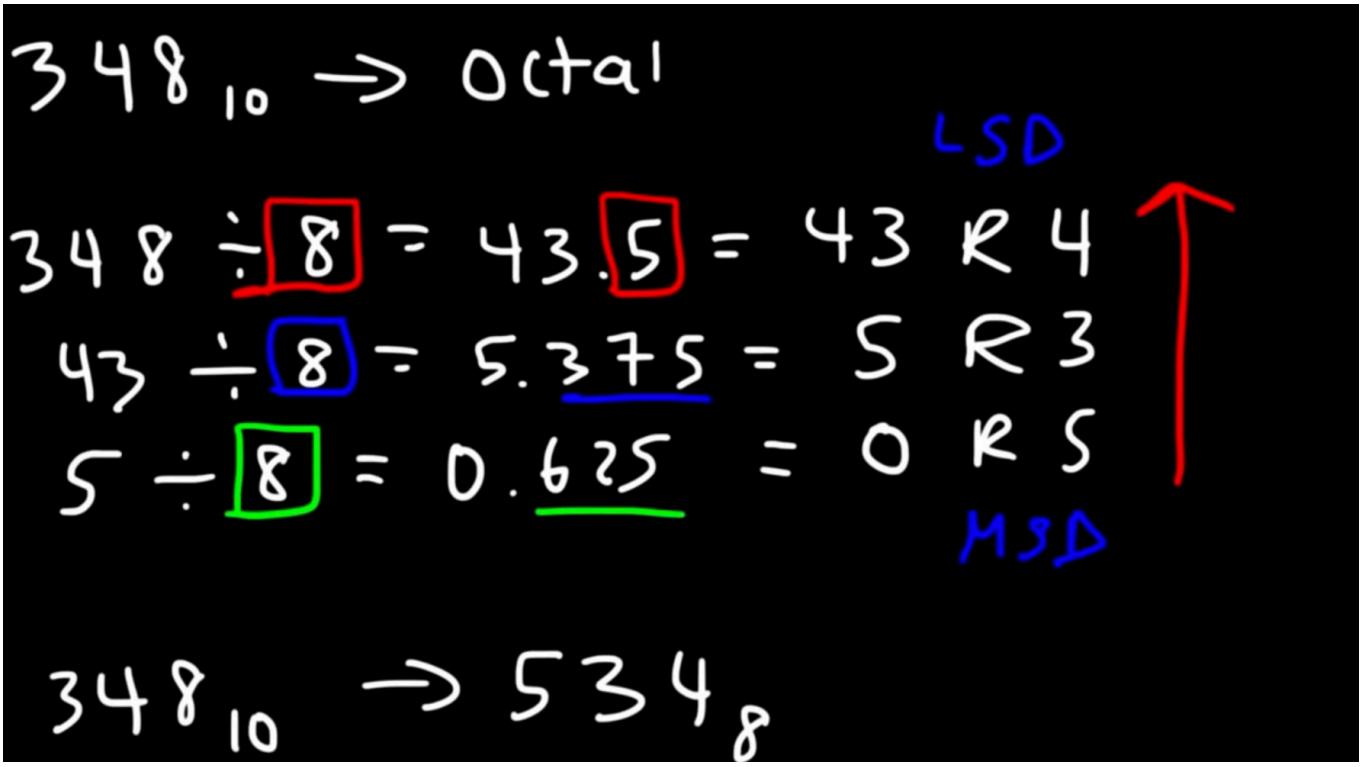
**Decimal to Binary**

- Via "Greedy Algorithm" (successive division)
  - find largest power of 2 that doesn't exceed number
  - subtract it from number
  - repeat until reach $2^0$

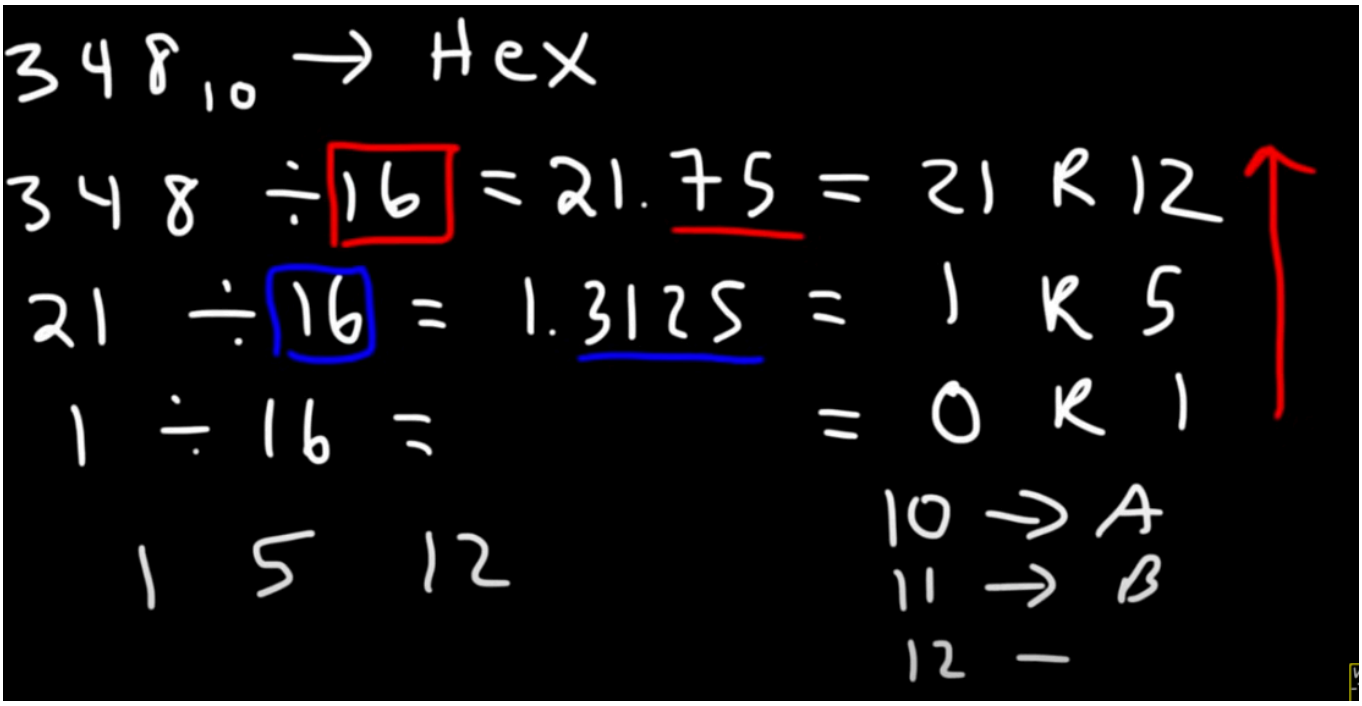



read from bottom to top (remainders will get you the bits - read from most significant to least significant bit )

**Decimal to Octal**

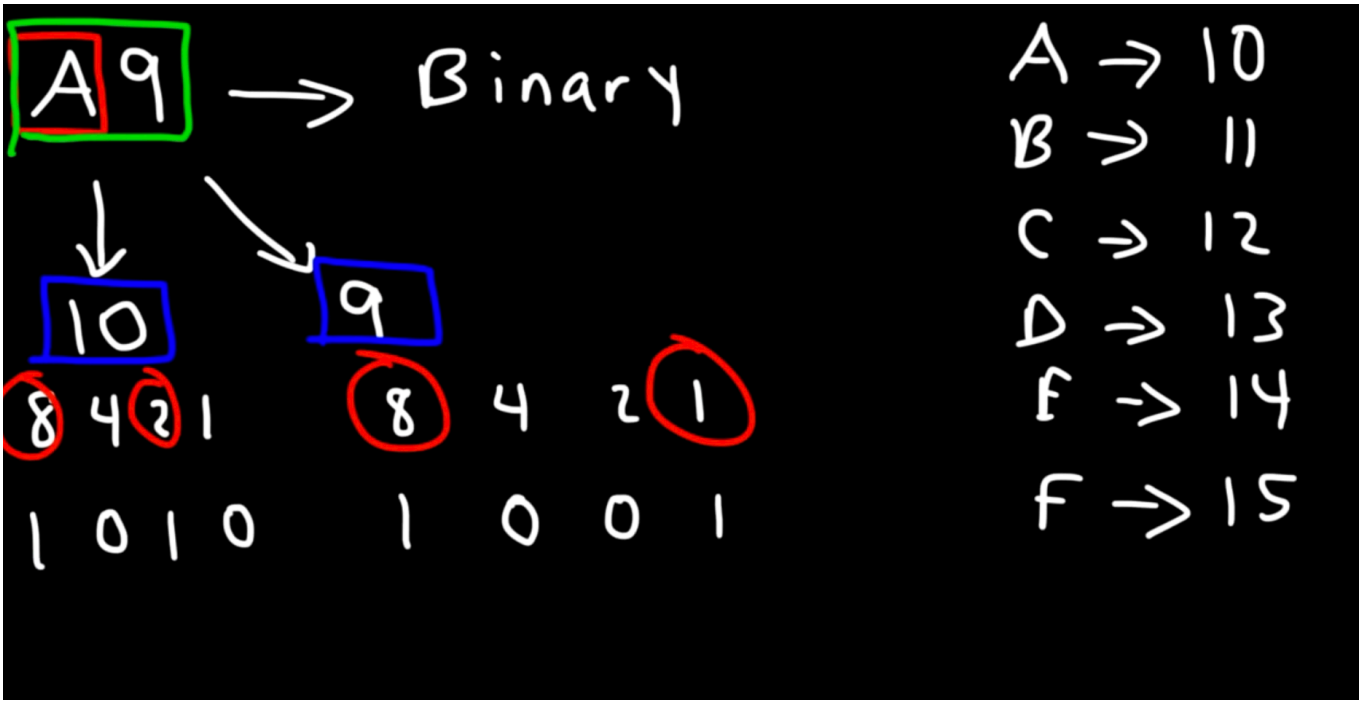



**Decimal to Hexadecimal**

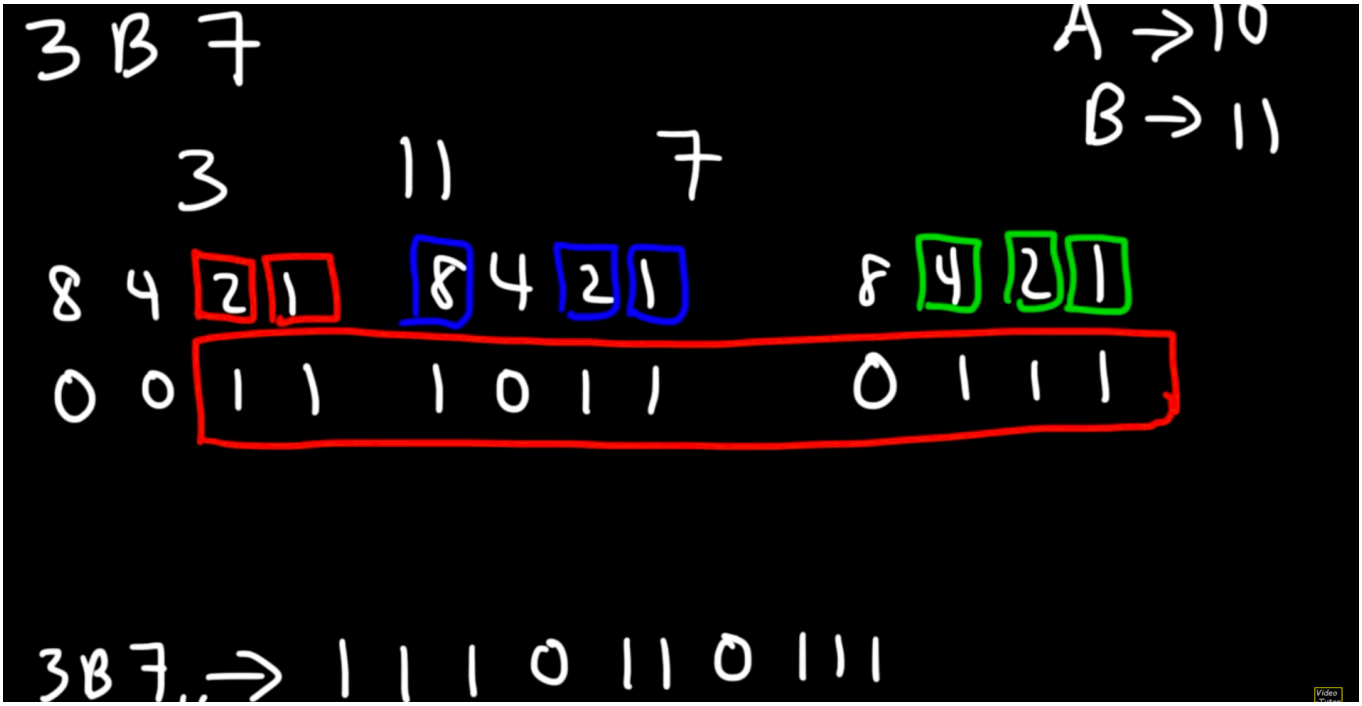



answer = 15C (as 12 is C in hexadecimal)

**Hexadecimal to Binary**




$A9_{16} = 10101001_2$ (0xA9 = 0b10101001)



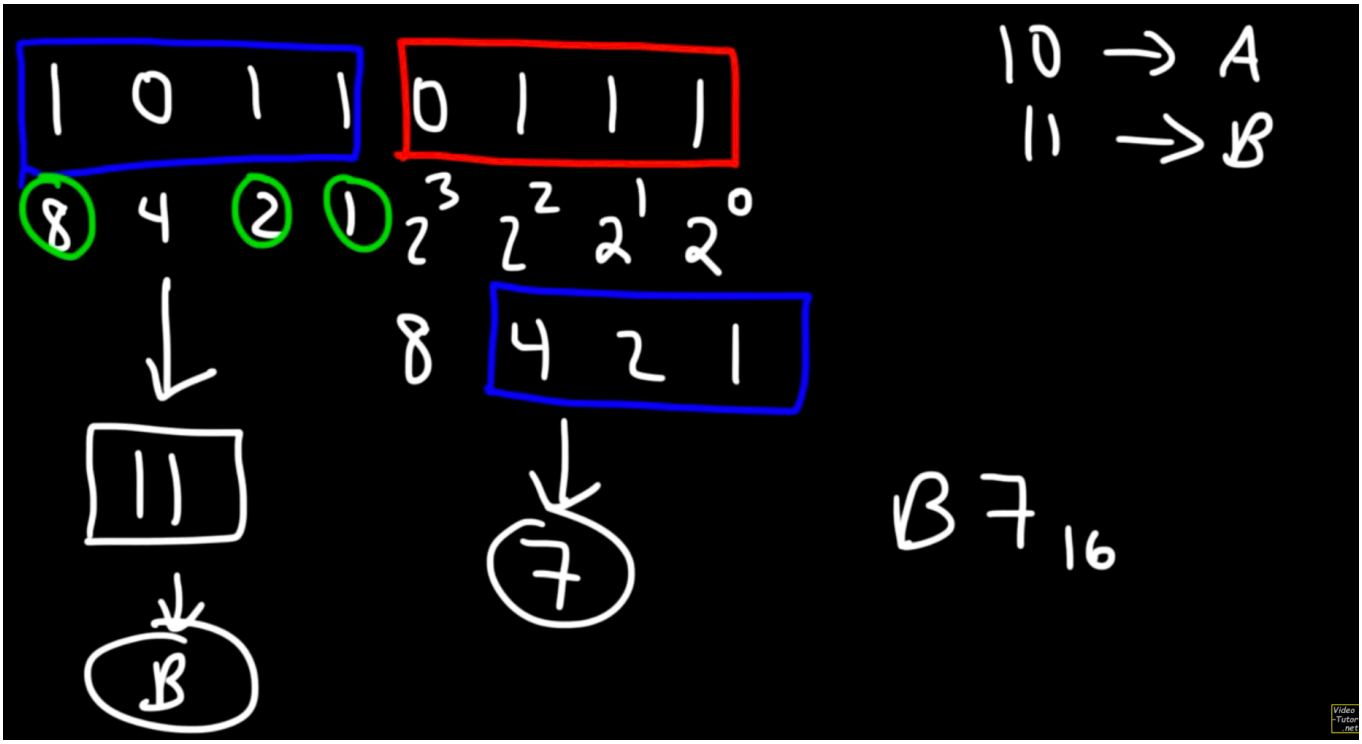

0x3B7 = 0b1110110111

(ignore leading zeros)

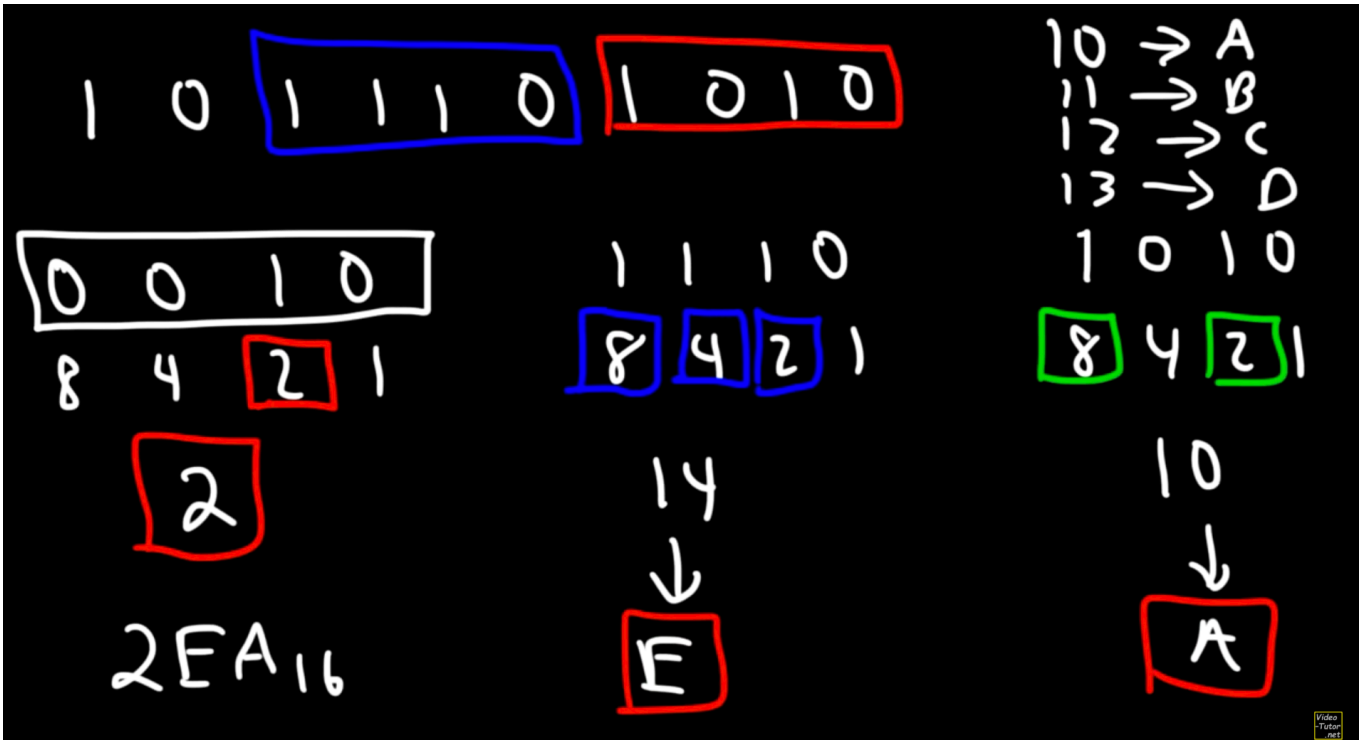
**Binary to Hexadecimal**

- separate binary numbers to groups of 4 (from end to start - remaining bits, just add leading zeroes to the left)
- convert to decimal value

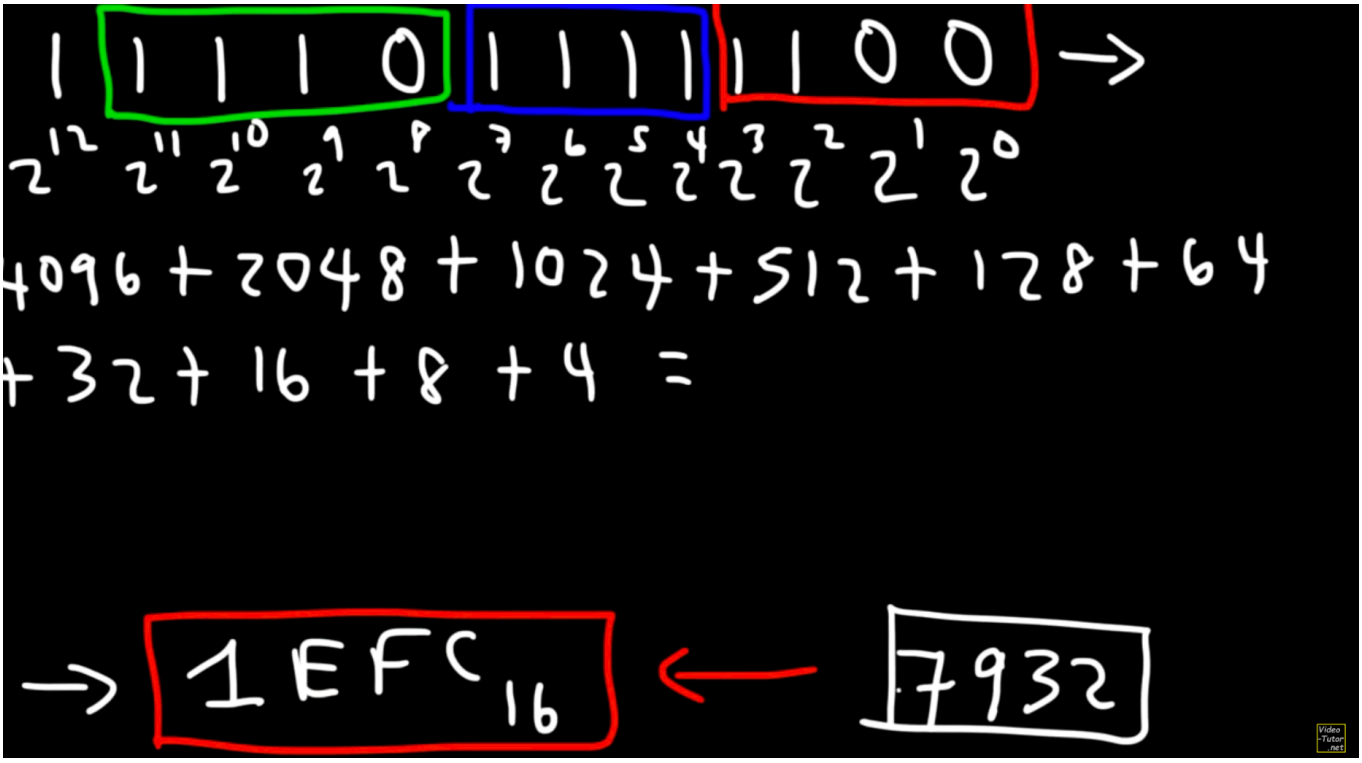
- convert to hexadecimal



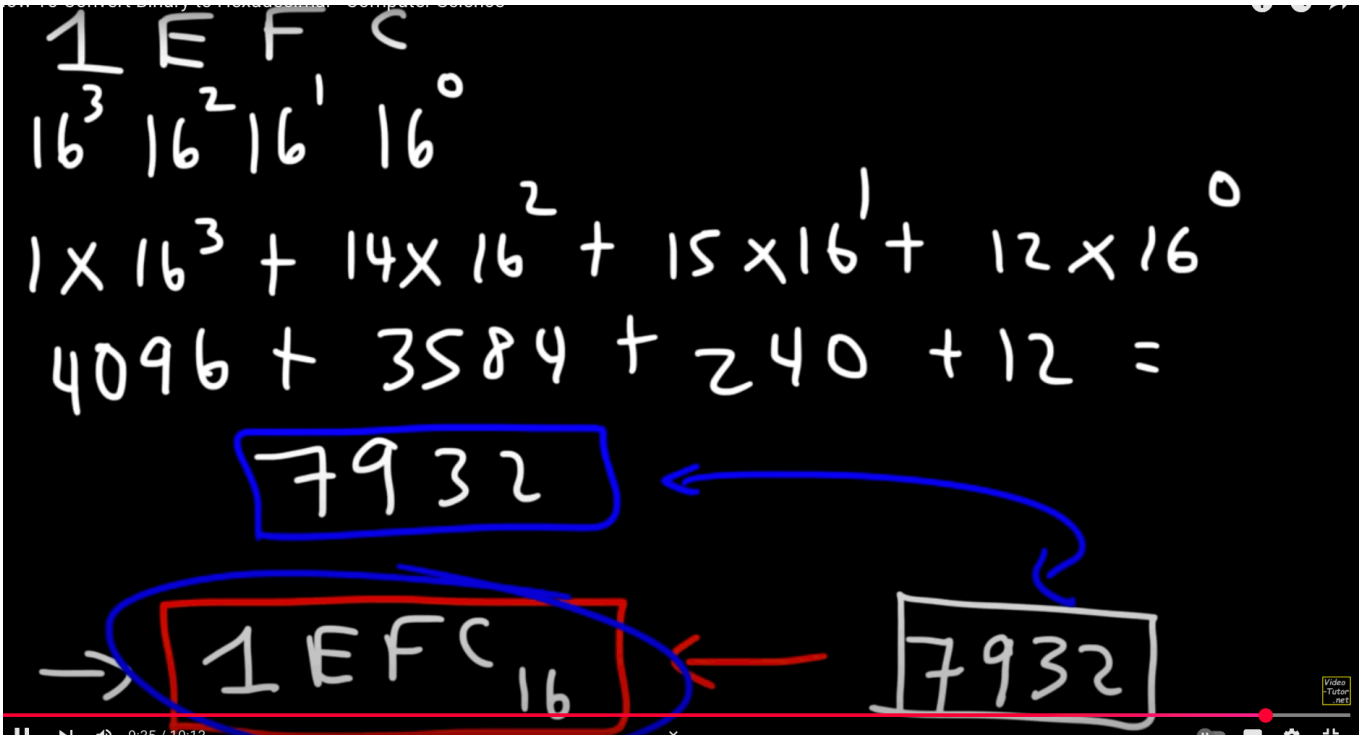

0b10110111 = 0xB7



**Binary to Decimal**




**Hexadecimal to Decimal**




# Notation of Number Systems

| | Base | Notation | Example | Range |
|---|---|---|---|---|
| Binary | 2 | 0b* | 0b11001 | 0 / 1 |

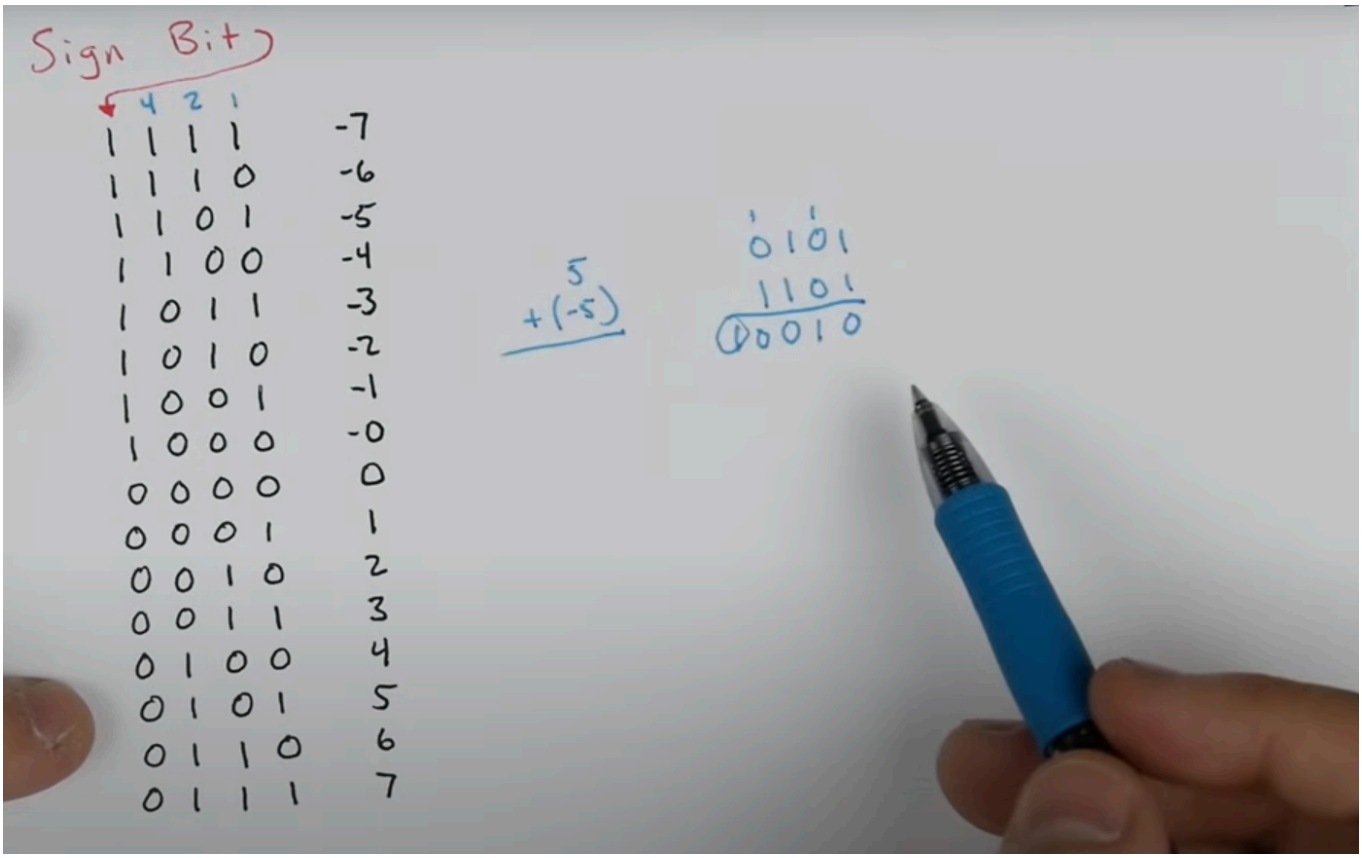
| | Base | Notation | Example | Range |
|---|---|---|---|---|
| Octal | 8 | 0* | 06037 | 01234567 |
| Decimal | 10 | NO prefix | 3123 | 0123456789 |
| Hexadecimal | 16 | 0x* | 0xFFE2 | 0123456789ABCDEF |

# Representing negative numbers

- Via **Two's Complement**
- (cannot use sign-magnitude representation to represent negative numbers in computers as 0 can have two representations (-0, +0)- not ideal, and that sign/magnitude can complicate calculations done by computers)
- computers can't subtract numbers, but can **add negative numbers**

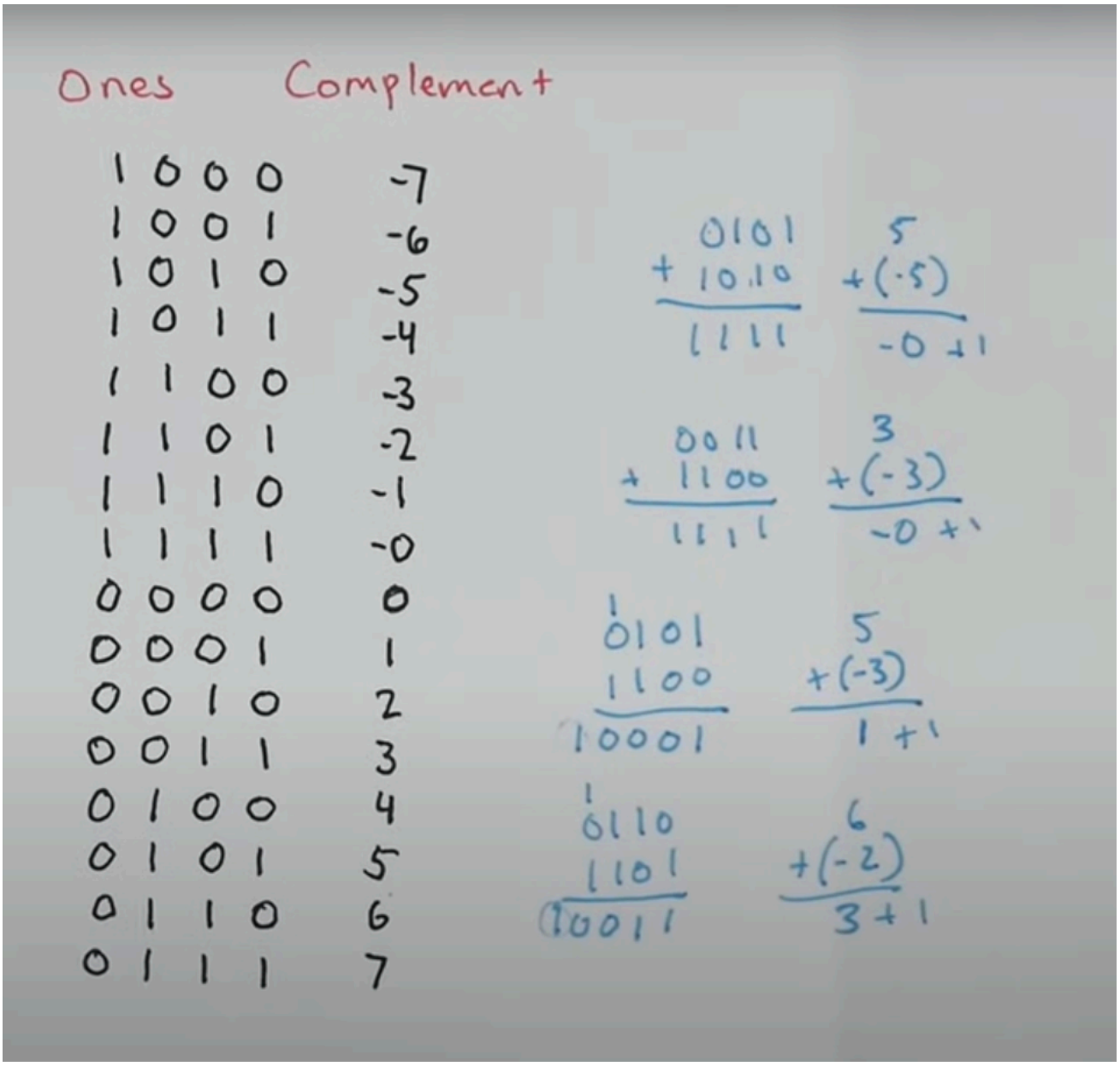
### Sign and Magnitude

- leftmost bit represents sign (1 = negative, 0 = positive)
- not easy to perform binary additions with hardware especially with overflow when binary addition when leftmost bit is signed - as its not part of the number but only tells the sign
- Can cause the two representation problem (-0 / +0) - cannot have 2 representations of zero.




### 1st Complement

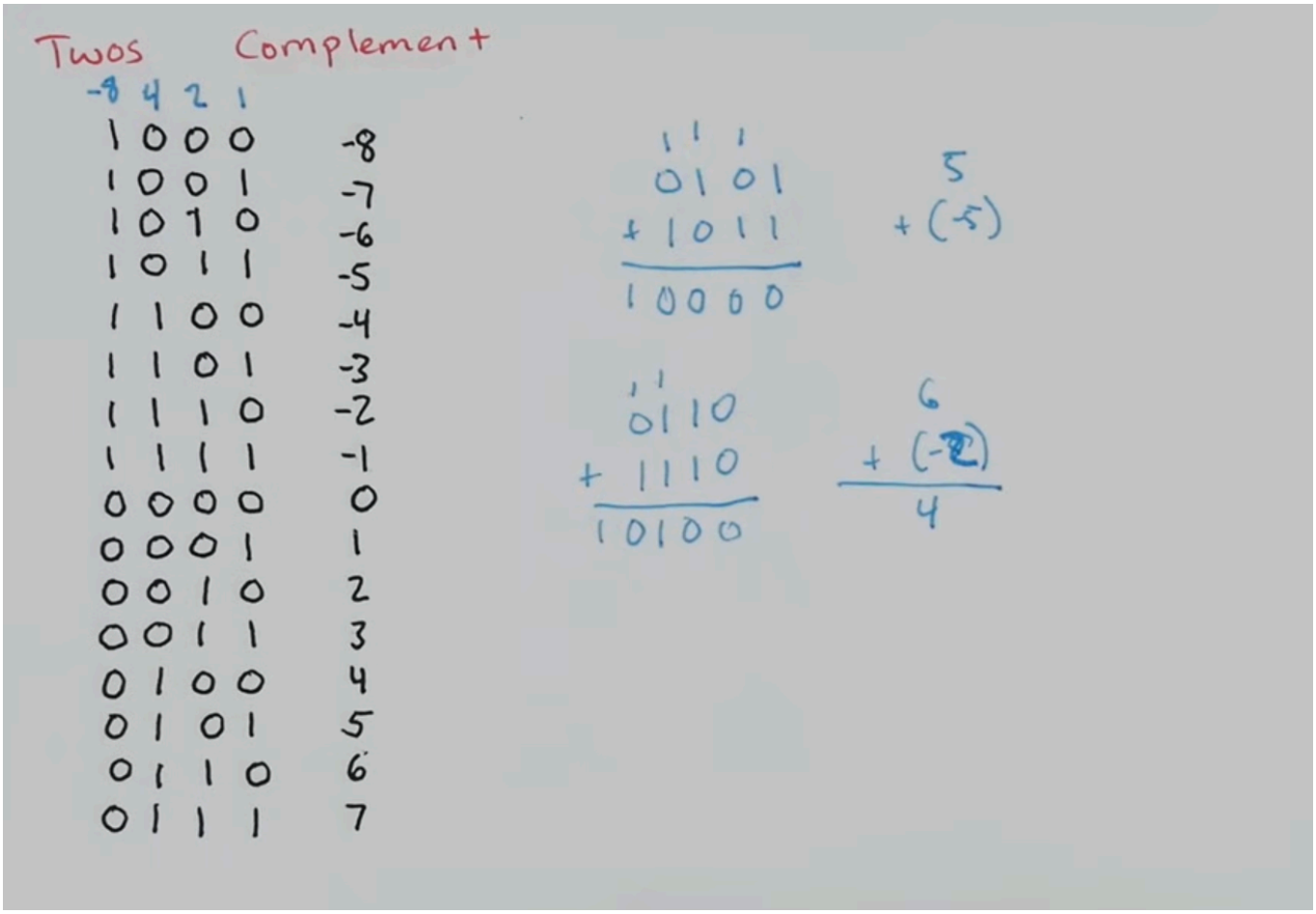
- Flip all the bits (1s become 0s, 0s become 1s) - hence the **complement**
- Two representation problem (-0 and +0) still exists
- Hardware cannot handle two representations of 0s
- Have to handle carry bits where you add the carry bit to start and perform addition again with carry bit as results can be off by 1 - not a good idea but it gets the answer




## Two's Complement

- **Equal to one's complement + 1**
- **Turn a value's binary representation into one's complement by flipping all its bits and then add one.**
- You shift the range by 1 -> helps to solve the two representation problem (there is no -0 representation) -> there is only 1 representation of 0 = easier for equalitty checking
- If there are any carry bits, just IGNORE them and leave them out

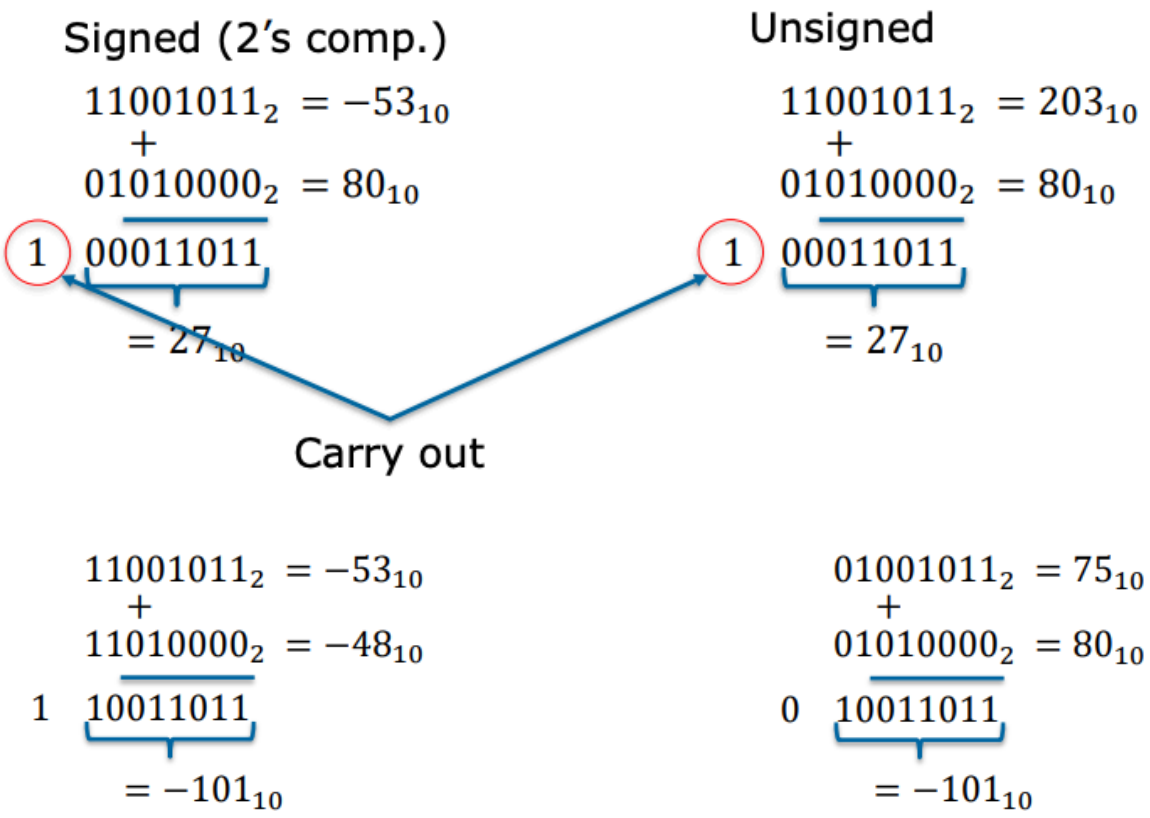
- allows numbers to be represented in the range [-2^n, 2^n - 1], where n is the number of bits
- most significant bit (leftmost) is represented as $-2^n$ where n is number of bits
- easier to perform binary arithmetic with two complement represented negative numbers (as computers' ALU only can perform binary addition but not binary subtraction)
- computers can perform both signed binary addition/multiplication while ignoring any carry bits




example: 0b10101 = -11 ($-2^4 + 2^2 + 2^0 = -16 + 4 + 1 = -11$)

## Overflow

- when adding two integers returns a larger number than can be represented (as we always ignore the carry out bit)
  - **pos + pos** and **neg + neg** can result in overflow
    - pos + pos can result in negative number being returned (if MSB answer gets 1 bit)
    - neg + neg can result in posiitve number being returned (if MSB answer gets 0 bit)
    - not what we want as pos + pos should return positive number and vice versa
  - **pos + neg** / **neg + pos** will never overflow







## Extending Numbers

- as leftmost bit (most significant bit) tells you the sign of the number,
- If positive number, just append 0s to left (MSB)
  - e..g. $00101010_2 \rightarrow 0000000000101010_2$
- If negative number, just append 1s to left (MSB)
  - e.g. $11001010_2 \rightarrow 1111111111001010_2$
- increase number of bits of binary number while preserving the number's sign and value

used when we want to perform binary arithmetic (keeping the same number of bits for both values)

e.g. 0b10101 = 0b110101

0b110101 = $-2^5 + 2^4 + 2^2 + 2^0 = -32 + 16 + 4 + 1 = -11$ (equivalent to 0b10101)

**overflow** = occurs when result of binary arithmetic cannot be represented (too big)

# Week 2 - Memory Units / MIPS Instructions

## Memory Units

### Integers

- Signed n-bit integers can represented within interval $[-2^{n-1}, 2^{n-1} - 1]$
  - e.g. 1 byte (8-bit) = $[-128, 127]$ (255 different numbers)
- Unsigned n-bit integers can be represented within interval $[0, 2^n - 1]$
  - e.g. 1-byte (8 bit) = $[0, 2^8 - 1] = [0, 255]$

### Memory Units

- Bit = 0 / 1 digit (binary digit)
- Byte = 8 bits (smallest addressable unit of memory)
- Kilobytes (KB) = 1024 bytes
  - its a power of 2 (its not equal to 1000 bytes - 1000 is SI prefix kilo)
- Megabytes (MB) = 1024 KB

### Words

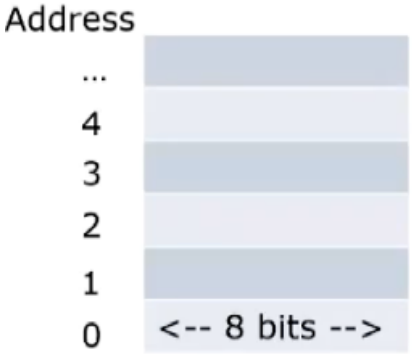
- Natural unit of data used by a given processor
- A word in **MIPS32** takes up 32 bits (as its a 32-bit processor)
- Indicates the size of
  - registers
  - memory addresses
  - instruction size
  - floating point variable sizes
- e.g. MIPS32 contain 32-bit registers/32-bit memory addresses/32 bit instructions/floating points are represented in 32 bits

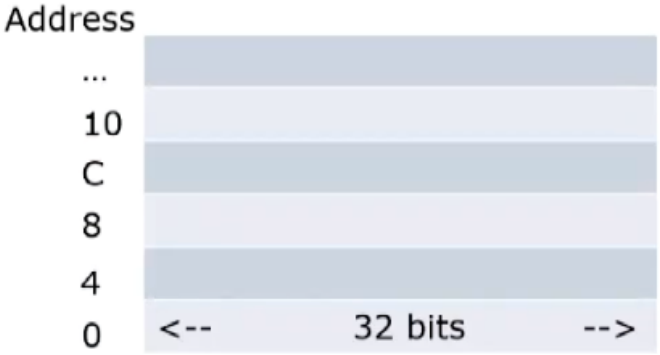
| Size in bits | Name (MIPS32) | Name (Intel x86) |
|---|---|---|
| 8 | Byte | Byte |
| 16 | Half word | Word |
| 32 | Word | Double word |
| 64 | Double word | Quad word |

### Memory Organisation

- Array of bytes (each location in array takes up 1 byte)
  - Each memory address identifies a single byte in memory
  - Each byte has its own memory address

    - Address of multi-byte object (e.g. word) is **lowest** address of all bytes it contains
- OR words (where every offset of 4 addresses can store 32 bits (1 word/4 bytes))
  - as word = 4 contiguous bytes




**Byte Order**

- **Big Endian**
  - Put MSB of the word at least address (smallest - lowest offset)
  - LSB is at largest memory address (highest offset)
  - **would require manipulation** of memory address (calculate offset) if want to convert e.g. from 32 bits to 16 bits
- **Little Endian**
  - Opposite order of big-endian
  - Put LSB of word at the smallest memory address (lowest offset)
  - MSB is at largest (highest offset)
  - Most computers store words in little-endian order
  - **no need for manipulation** of memory address (calculation of offset) if want to convert e.g. 32 bits to 16 bits

- Consider the word 0xBABACAFE.
  - How'd we store it in address 8?

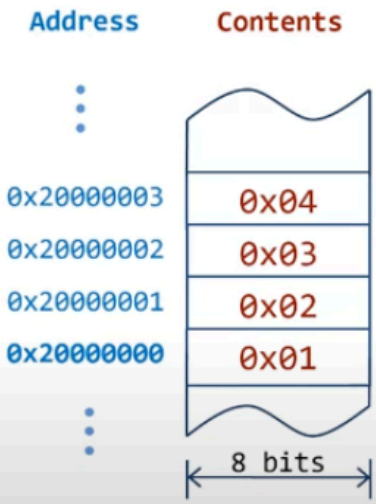
| Address | Offset 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| ... | | | | |
| 10 | | | | |
| C | | | | |
| 8 | BA | BA | CA | FE |
| 4 | 0 | 0 | 0 | 9 |
| 0 | | | | |

Big Endian

| Address | Offset 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| ... | | | | |
| 10 | | | | |
| C | | | | |
| 8 | FE | CA | BA | BA |
| 4 | 9 | 0 | 0 | 0 |
| 0 | | | | |

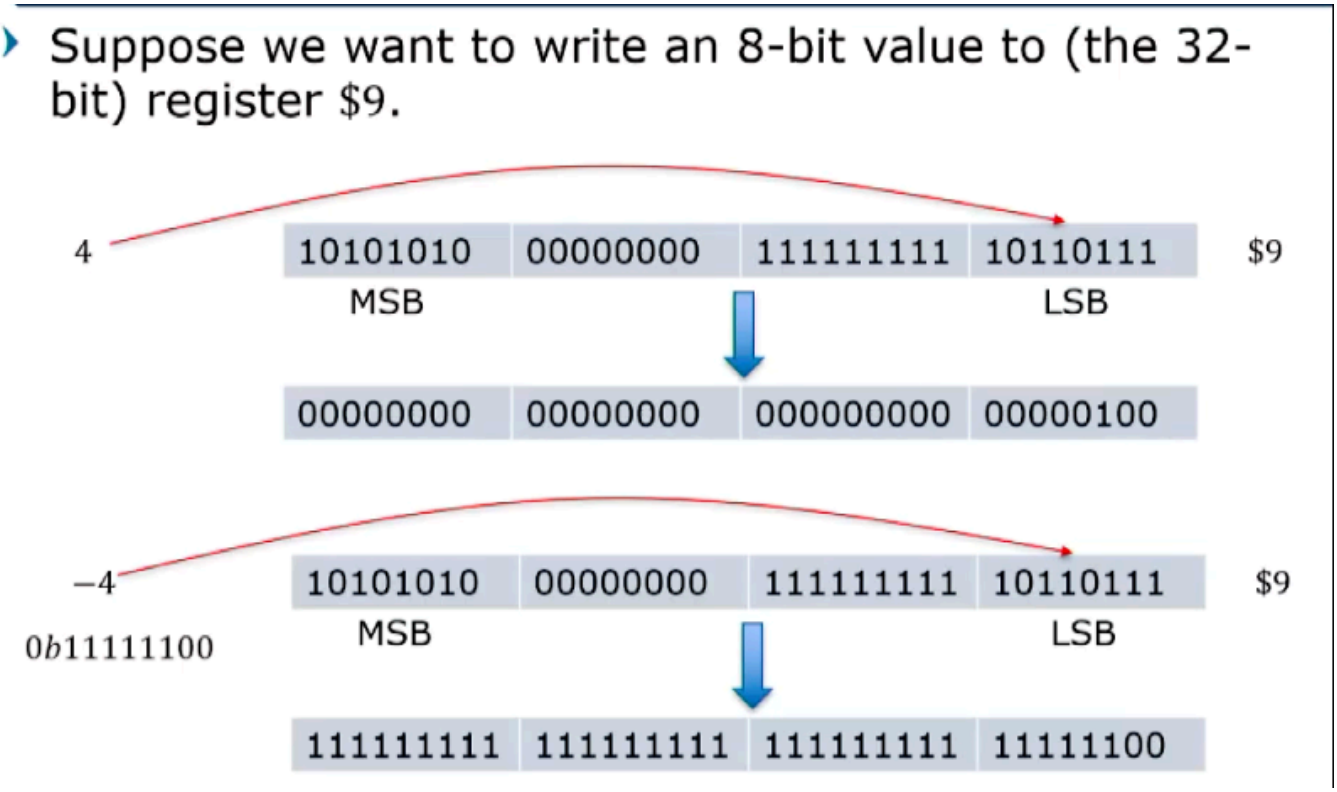
Little Endian

- Say that the 32-bit integer 9 is stored in address 4




- If Little Endian
  value = 0x04030201
- If Big Endian
  value = 0x01020304

**How to set register values**




- You'll need to sign extend two complement (negative numbers) if want to write negative numbers to a register to 32 bits

# MIPS Instructions

Instructions are written in 32 bits in MIPS32 - include opcode/operands whre instructions are represented as numbers.
Note; MIPS is a RISC (Reduced Instruction Set Computing) processor

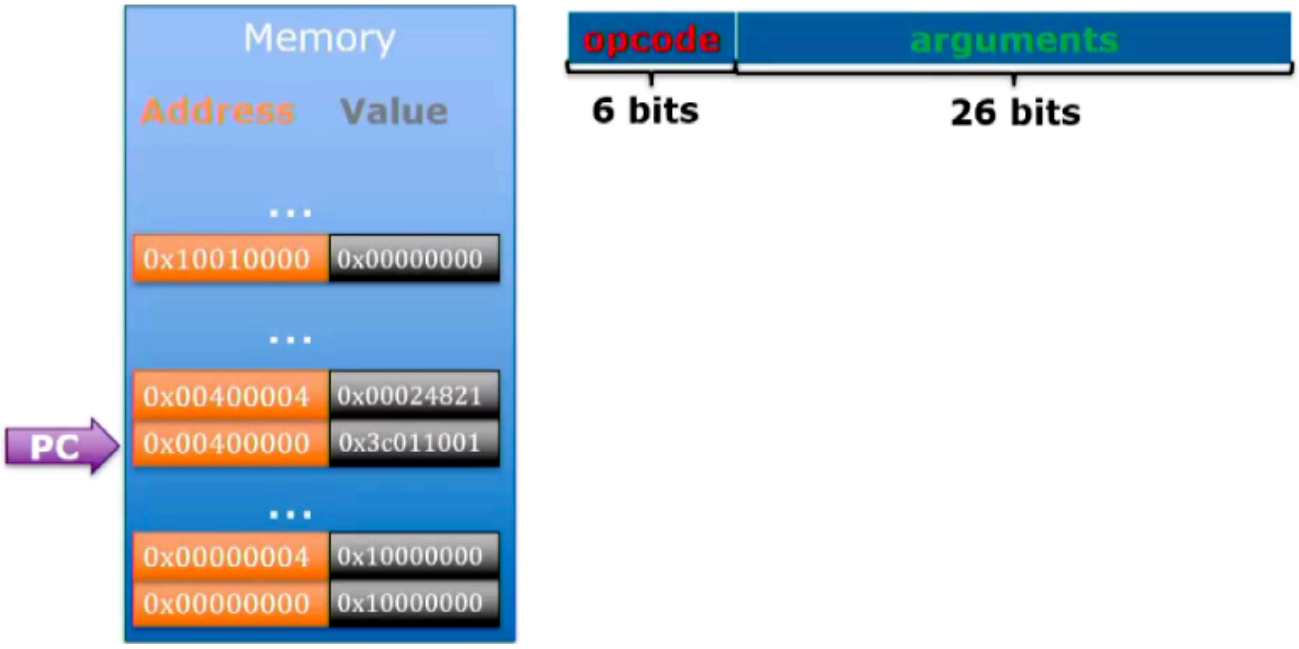
**MIPS Registers**

- MIPS32 has 32 registers and memory addresses are 32 bits long
- Different registers have different purposes e.g. $0 will always have value 0
- MIPS only operate on registers (cannot perform operations with only memory addresses)
- E.g. `Mem[0x1001000] += Mem[0x1001004]`
  - read `Mem[0x1001000]` into register
  - Read `Mem[0x1001004]` into another register
  - Compute sum of both register
  - Store result into `Mem[0x1001000]`

**MIPS PRocessors**

MIPS have simple ISA (instruction set architecture) which contain 32-bit instructions (refer to

the reference card)




- Processor has the program counter which points at the address of the instruction to be executed
  - Take the value of the address (it contains the instructions)
  - Decode it
    - Opcode (first 6 bits) = Tells what type of instruction
    - Arguments (remaining 26 bits) = depends on the instruction (specifically the opcode)
  - Move the program counter to point to next instruction (increment by 4 as each address stores 8 bits, and that instructions are 32 bits, thus an offset of 4)

**Instruction Types**
Use the Reference Card to see which operation to run, as well as their arguments and function number to see how these instructions are written in machine code (stored in memory).

- Reference card provides an overview of the most important MIPS instructions but Appendix A of patterson/hennesey book provides a comprehensive reference of all MIPS instructions

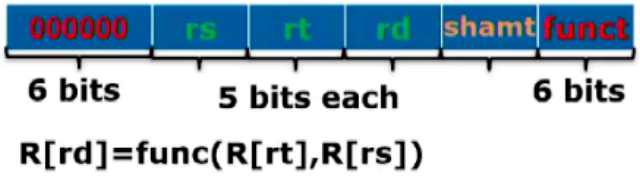
Looking at the opcode (first 6 bits of the instruction) to see which type of operation e.g. opcode 0 is a rithmetic,

- **Rithmetic** (between register) operations
  - just specify register numbers only (**they have an opcode of 0**. Function number distinguish the different operations)
  - **op** = operation of instruction aka opcode, **rs** = first register source operand, **rt** = second register source operand, **rd** = register destination operand, **shamt** = shift

amount (normally zero for non-shift instructions), **funct** = function code (specific variant of operation on the opcode field)

- e.g. `add $8, $1, $2`
  - add the values stored in register 1 and register 2
  - write the sum into register 8
    - Opcode is 0.

| NAME, MNEMONIC | | FOR-MAT | OPERATION (in Verilog) | / FUNCT (Hex) |
|---|---|---|---|---|
| Add | add | R | R[rd] = R[rs] + R[rt] | (1) $0 / 20_{hex}$ |



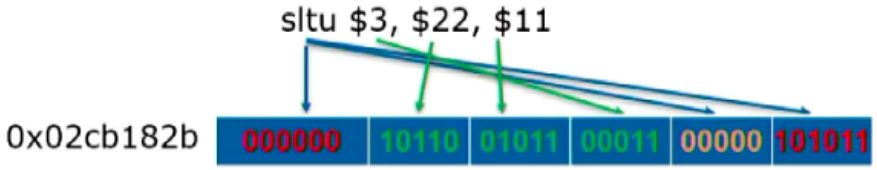


  - by referencing the MIPS Reference Sheet (Green Card) - translation of that instruction to machine code (the instruction stored in memory)
- e.g. `sub $12, $6, $3`
  - take the difference between register 6 and register 3
  - write the difference into register 12


| | | | | |
|---|---|---|---|---|
| Subtract | sub | R | R[rd] = R[rs] - R[rt] | (1) $0 / 22_{hex}$ |

  - opcode = 00000
  - the last 6 bits represent the function number (shown as 22_hex in example - 100010 in binary = 34 in decimal)
- e.g. `sltu $3, $22, $11`
  - Compare and see if value stored in register 22 is less than value stored in register 11.
  - If value is lesser, set value in register 3 to 1, otherwise set to 0

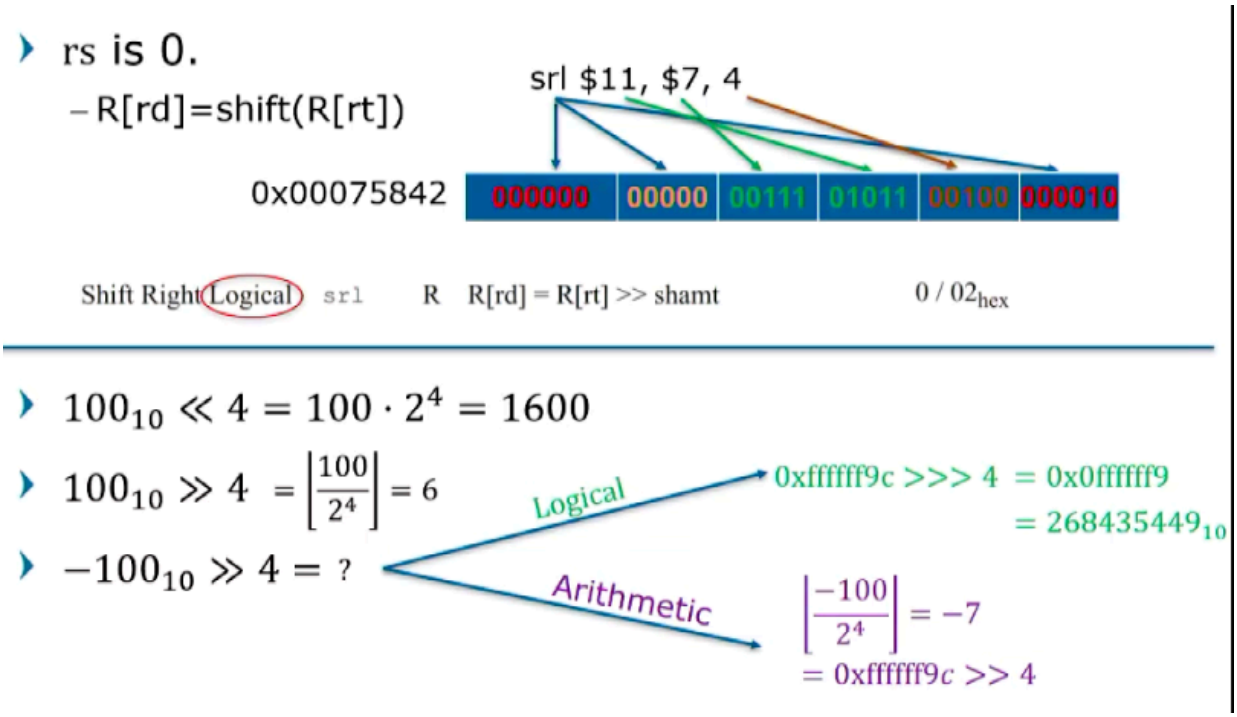
Set Less Than Unsig. `sltu` R R[rd] = (R[rs] < R[rt]) ? 1 : 0 (6) $0 / 2b_{hex}$

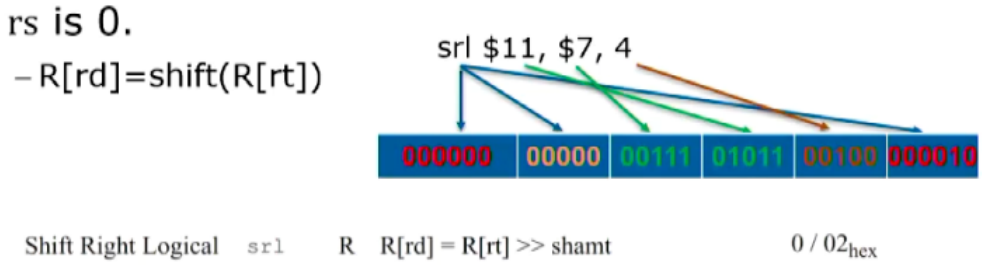



- Example of Rithmetic instructions

| Instruction | op | funct (Base 16) | Description |
|---|---|---|---|
| **add** | 0 | 0x20 | Addition – with signed overflow |
| **sub** | 0 | 0x22 | Subtraction – with signed overflow |
| **and** | 0 | 0x24 | Logical And |
| **or** | 0 | 0x25 | Logical Or |
| **xor** | 0 | 0x26 | Logical Xor |
| **nor** | 0 | 0x27 | Logical Nor |
| **slt** | 0 | 0x2A | Set on less than |
| **sltu** | 0 | 0x2B | Set on less than unsigned |
| **srl** | 0 | 0x02 | Shift Right Logical |
| **sra** | 0 | 0x03 | Shift Right Arith. |
| **sll** | 0 | 0x00 | Shift Left Logical |

- **Immediate** (including fixed numbers) operations:
  - contains a value encoded directly within the instruction (on a constant)
  - e.g. `addi $15, $15, -1`
    - subtract 1 (or add -1) from the value stored in register 15
    - write the result into register 15
  - e.g. `beq $10, $14, 0x00000002`
    - check if values stored in register 10 and register 14 are the same (branch equals)
    - if they are equal, branch (change value of program counter to `0x00000002` = change what is the next instruction to execute)
- **Jump** operations
  - Allows the program to jump out of one section of program into another
  - e.g. `j 0x00400050`
    - Jump to address `0x00400050`
  - e.g. `jal 0x0040007c`
    - Jump and link address `0x0040007c`
- **Shift** instructions

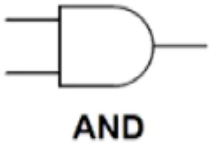



- **Logical shifts**
  - **Shift right logical** ( `srl` in MIPS)
    - **Remove the n least significant bits and add n number of bits to most significant bits** (same as dividing by $2^n$ - integer division)
    - treats all bits equally, including sign bit (use when unsigned)
    - e.g. $100_{10} >>> 4 = \frac{100}{2^4} = 6$
    - 

    - shift value in register 7 by 4 and store result in register 11
  - **Shift left logical** ( `sll` in MIPS)
    - **Shift all bits to left and append n number of 0s to the end** - most signifnicant bits (same as multiplying by $2^n$)
    - e.g. $100_{10} << 4 = 100 \times 2^4 = 1600$
    - $1100100_2 >> 4 = 11001000000_2$
- **Arithmetic shifts**
  - Left shift (<<) - can't find the mips equiv
    - Identical to logical shift left
  - Right shift (>>) - `sra` in MIPS32

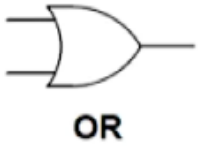
    - shifts all bits to right but preserves sign bit
        - fill with 1s for negative
        - fill with 0s for positive
    - use when data is signed
    - e.g. $0\text{xffffff9c} >> 4 = 0\text{xfffffff9}$

- **Bitwise operations**
    - Apply bit by bit (bitwise)



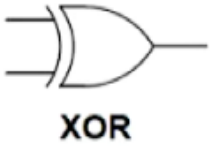
**AND**

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



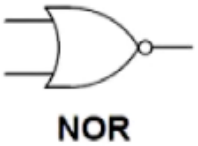
**OR**

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



**XOR**

| A | B | Output |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



**NOR**

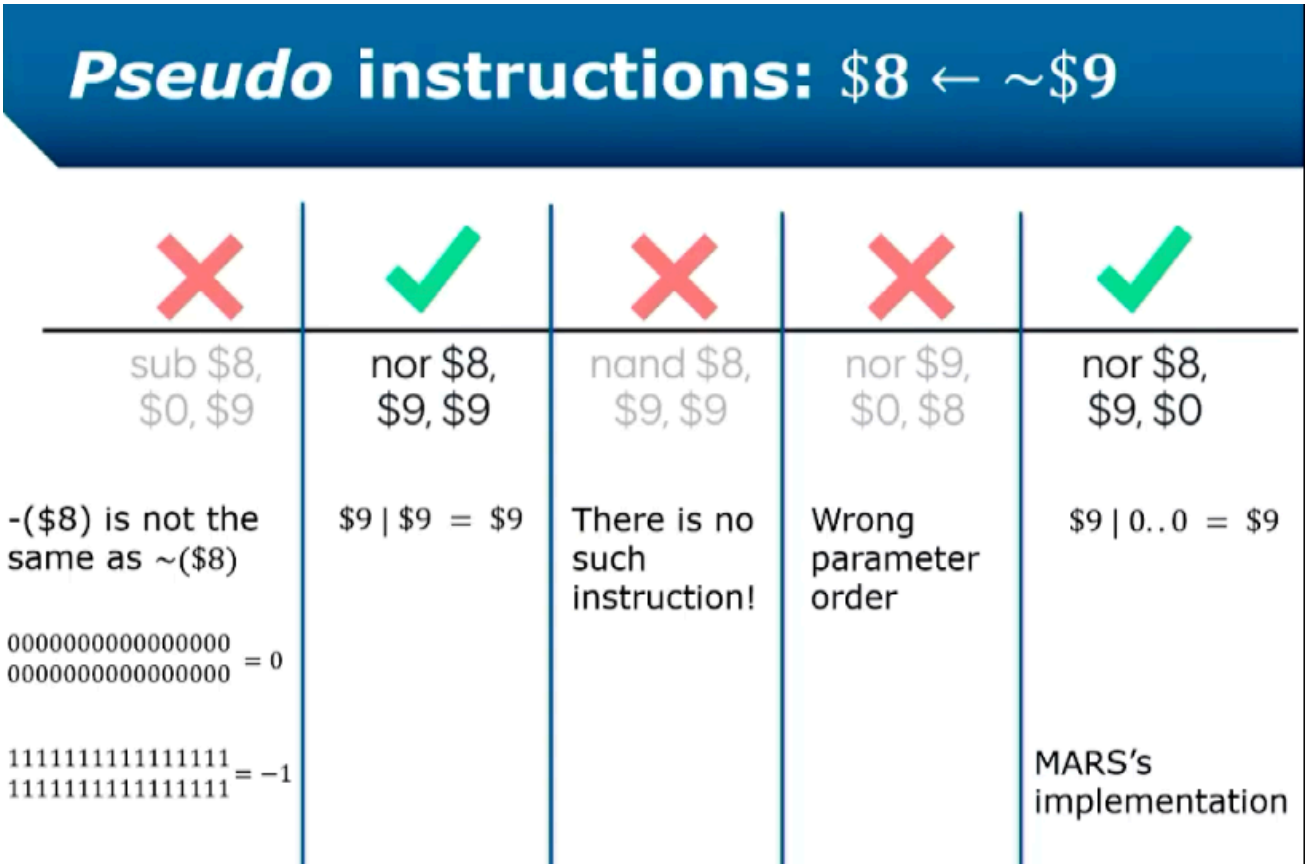
| A | B | Output |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

    - Bitwise AND
    - e.g

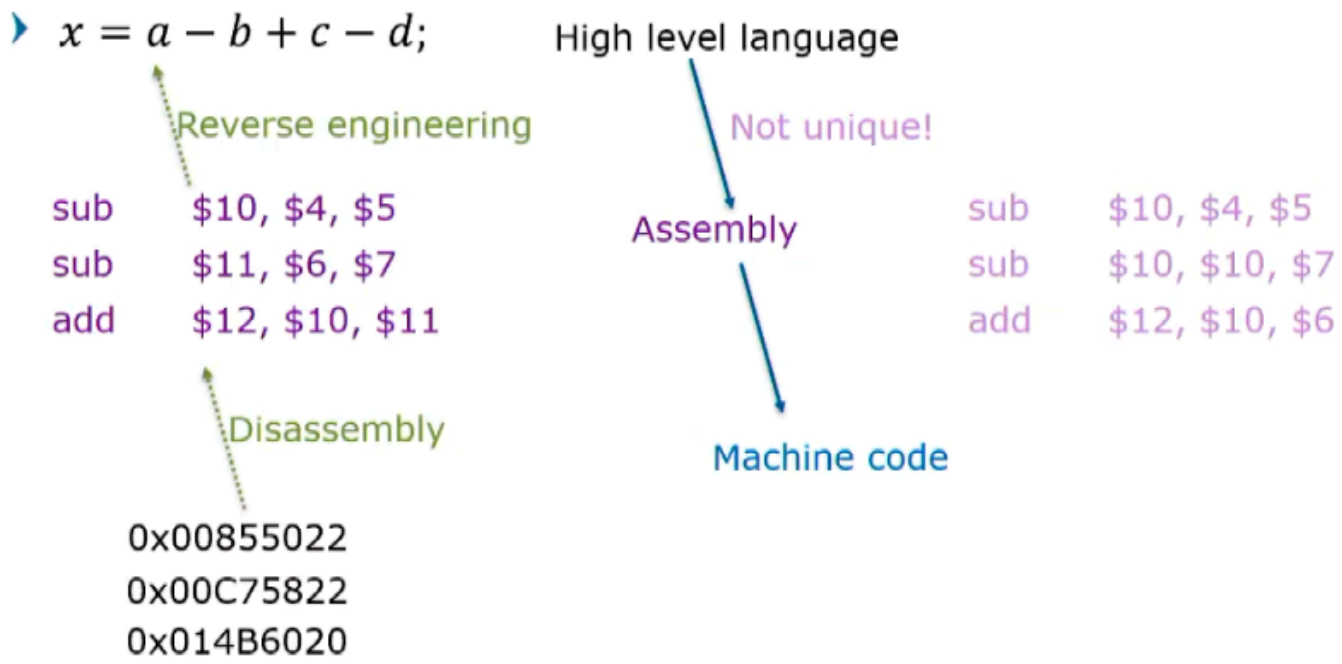
**Pseudo Instructions**

- Can be written in Assembly
- not real MIPS32 instructions
    - dont have opcodes / unrecognised by CPU
- are translated to real MIPS32 instructions by the assembler
- because there is limited number of opcodes (only 6 bits in instructions are reserved for opcodes)
- e.g. pseudo instruction `move $8, $9` will be translated into `add $8, $9, $0`
  - where move means you store what is in register 9 as register 8
  - in real mips32 code, you just add 0 (register $0 will always store 0 no matter what operation you try to do) to what is stored in register 9 and store result in register 8 (baiscally the same as assigning the value of register 9 into register 8)

example:




**High-level commands**




Compilation involves converting high level code to efficient assembly code which is then converted to unique machine code

There is only 1 way to map MIPS instructions to machine code

**Exam Question**

Q1a. Which pair of instructions always has an identical effect (i.e., the first can be replaced by the second and vice versa), regardless of the initial register values?

[(-2,+5) marks]

A) `add $t0, $t1, $t1` , `addu $t0, $t1, $t1`
B) `add $t0, $t1, $t1` , `sll $t0, $t1, 1`
C) `addu $t0, $t1, $t1` , `srl $t0, $t1, 1`
D) `addu $t0, $t1, $t1` , `sll $t0, $t1, 1`

a) wrong. as add can cause overflow exception if adding two large negative numbers
b) wrong, add can cause overlfow exception (sll doesnt). sll doubles the number
c) srl means dividing by 2 (you half the number, not make the number greater)
**d) correct, as adding the same number twice is same as doubling number (sll)** and that both instructions do not throw any exceptions

You can check the reference card to see if the instructions can throw exceptions, and see what these instructions do (to be able to determine your answer).

Q1b. The instruction `add $t0, $t1, $t1` would never throw an exception if:

[(-2,+5) marks]

A) $t1 has a positive value.
B) $t1 has a negative value.
C) The previous instruction was: `sra $t1, $t1, 1`.
D) The previous instruction was: `srl $t1, $t1, 1`.

a) false, too large numbers can throw overflow exception
b) false, too large negative numbers can throw an overflow exception
**c) true, `sra` = shift right arithmetic considers the sign bit (add 1/0 to MSB depending on sign), so addition can cause overflow - especially when performing `sra` on negative numbers and adding them as 1+1 for MSB will cause overflow, as sra will make the MSB 1 of the result for negative numbers. however sra won't throw any exception**
d) true, `srl` = shift by logical disregards the sign bit as it will always add 0 to the most signifcant bit, so adding by itself won't get you overflow (as compared to `sra`). counter example is that D will throw an exception if $t1 was -1

# Week 3 - MIPS Instructions (continued) / Combining MIPS Instructions

# MIPS Instructions (Continued)

## Background Reading - 2.3 (Computer Organization and Design)

Every computer needs to know how to perform arithmetic, including MIPS

### Example of operands/assembly language code

**MIPS operands**

| Name | Example | Comments |
|---|---|---|
| 32 registers | `$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at` | Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants. |
| $2^{30}$ memory words | Memory[0], Memory[4], . . . , Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers. |

**MIPS assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | `add $s1,$s2,$s3` | $s1 = $s2 + $s3 | Three register operands |
| | subtract | `sub $s1,$s2,$s3` | $s1 = $s2 – $s3 | Three register operands |
| | add immediate | `addi $s1,$s2,20` | $s1 = $s2 + 20 | Used to add constants |
| Data transfer | load word | `lw $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Word from memory to register |
| | store word | `sw $s1,20($s2)` | Memory[$s2 + 20] = $s1 | Word from register to memory |
| | load half | `lh $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Halfword memory to register |
| | load half unsigned | `lhu $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Halfword memory to register |
| | store half | `sh $s1,20($s2)` | Memory[$s2 + 20] = $s1 | Halfword register to memory |
| | load byte | `lb $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Byte from memory to register |
| | load byte unsigned | `lbu $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Byte from memory to register |
| | store byte | `sb $s1,20($s2)` | Memory[$s2 + 20] = $s1 | Byte from register to memory |
| | load linked word | `ll $s1,20($s2)` | $s1 = Memory[$s2 + 20] | Load word as 1st half of atomic swap |
| | store condition. word | `sc $s1,20($s2)` | Memory[$s2+20]=$s1;$s1=0 or 1 | Store word as 2nd half of atomic swap |
| | load upper immed. | `lui $s1,20` | $s1 = 20 * $2^{16}$ | Loads constant in upper 16 bits |
| Logical | and | `and $s1,$s2,$s3` | $s1 = $s2 & $s3 | Three reg. operands; bit-by-bit AND |
| | or | `or $s1,$s2,$s3` | $s1 = $s2 \| $s3 | Three reg. operands; bit-by-bit OR |
| | nor | `nor $s1,$s2,$s3` | $s1 = ~ ($s2 \| $s3) | Three reg. operands; bit-by-bit NOR |
| | and immediate | `andi $s1,$s2,20` | $s1 = $s2 & 20 | Bit-by-bit AND reg with constant |
| | or immediate | `ori $s1,$s2,20` | $s1 = $s2 \| 20 | Bit-by-bit OR reg with constant |
| | shift left logical | `sll $s1,$s2,10` | $s1 = $s2 << 10 | Shift left by constant |
| | shift right logical | `srl $s1,$s2,10` | $s1 = $s2 >> 10 | Shift right by constant |
| Conditional branch | branch on equal | `beq $s1,$s2,25` | if ($s1 == $s2) go to PC + 4 + 100 | Equal test; PC-relative branch |
| | branch on not equal | `bne $s1,$s2,25` | if ($s1!= $s2) go to PC + 4 + 100 | Not equal test; PC-relative |
| | set on less than | `slt $s1,$s2,$s3` | if ($s2 < $s3) $s1 = 1; else $s1 = 0 | Compare less than; for `beq`, `bne` |
| | set on less than unsigned | `sltu $s1,$s2,$s3` | if ($s2 < $s3) $s1 = 1; else $s1 = 0 | Compare less than unsigned |
| | set less than immediate | `slti $s1,$s2,20` | if ($s2 < 20) $s1 = 1; else $s1 = 0 | Compare less than constant |
| | set less than immediate unsigned | `sltiu $s1,$s2,20` | if ($s2 < 20) $s1 = 1; else $s1 = 0 | Compare less than constant unsigned |
| Unconditional jump | jump | `j 2500` | go to 10000 | Jump to target address |
| | jump register | `jr $ra` | go to $ra | For switch, procedure return |
| | jump and link | `jal 2500` | $ra = PC + 4; go to 10000 | For procedure call |

When translating C code to MIPS, one line C statement can be broken into several assembly instrctions e.g. `f = (g+h) – (i+j)` would be

```
add $t0, g, h # t0 = g + h
add $t1, i, j # t1 = i + j
sub f, $t0, $t1 #t0 – t1 = (g + h) – (i + j)
```

where we declare tempoary variables

## 2.3 - Operands of Computer Hardware

Operands of arithmetic instructions are restricted - they must be from limited number of special locations built directly in hardware called **registers** (MIPS instructions mainly use registers, apart from immediate/jump instructions)

Size of registers in MIPS architecture are 32 bits and groups of 32 bits occur frequently, giving the name **word** (natural unit of access in a computer, usually a group of 32 bits; corresponds to size of register in MIPS architecture)

The 3 operands of MIPS arithmetic instructions must each be chosen from one of the 32 32-bit registers

Why 32 registers?

- design principle - smaller is faster (to keep the clock cycle fast)

**Compiling a C Assignment Using Registers**

EXAMPLE

It is the compiler's job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

```
f = (g + h) - (i + j);
```

The variables `f`, `g`, `h`, `i`, and `j` are assigned to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively. What is the compiled MIPS code?

The compiled program is very similar to the prior example, except we replace the variables with the register names mentioned above plus two temporary registers, `$t0` and `$t1`, which correspond to the temporary variables above:

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

**Memory Operands**

Note that data structures can be stored in memory, but processor can only keep small amount of data in registers. How do computers represent and access large, complex data structures that can be larger than what can be stored by registers?

MIPS have **data transfer instructions** = instructions that transfer data between memory and registers, as arithmetic operations can only occur on registers in MIPS.

How to access a word in memory?

- instruction must supply memory address (where memory is just a large 1d darray with address acting as index to that array, starting at 0)
- e.g. address of 3rd data element is 2 and value of Memory[2] = 10 (the data value stored in address 2)
- MIPS uses **byte addresing** where 1 word represents 4 byytes\
- data transfer instruction that copies data from memory to register is called load, where format of load instruction is name of operation followed by register to be loaded and then constatn and register used to access memory.
- memory address is formed by summing constant portion of the instruction and the contents of the second register
- to load a word in MIPS to a register, you'll use the instruction `lw` (load word)

example:

**Compiling an Assignment When an Operand Is in Memory**

**EXAMPLE**

Let's assume that `A` is an array of 100 words and that the compiler has associated the variables `g` and `h` with the registers `$s1` and `$s2` as before. Let's also assume that the starting address, or *base address*, of the array is in `$s3`. Compile this C assignment statement:

```
g = h + A[8];
```

**ANSWER**

Although there is a single operation in this assignment statement, one of the operands is in memory, so we must first transfer `A[8]` to a register. The address of this array element is the sum of the base of the array `A`, found in register `$s3`, plus the number to select element 8. The data should be placed in a temporary register for use in the next instruction. Based on Figure 2.2, the first compiled instruction is

```
lw   $t0,8($s3) # Temporary reg $t0 gets A[8]
```

(On the next page we'll make a slight adjustment to this instruction, but we'll use this simplified version for now.) The following instruction can operate on the value in `$t0` (which equals `A[8]`) since it is in a register. The instruction must add `h` (contained in `$s2`) to `A[8]` (`$t0`) and put the sum in the register corresponding to `g` (associated with `$s1`):

```
add  $s1,$s2,$t0 # g = h + A[8]
```

The constant in a data transfer instruction (8) is called the *offset*, and the register added to form the address (`$s3`) is called the *base register*.

compiler will allocate data structures e.g. arrays and structures to locations in memory and then can place proper starting address into data transfer instructions

the address of a word matches address of one of the 4 bytes within the word,a nd that addresses of sequential words differ by 4. In MIPS, words must start at addresses that are multiples of 4 - **alignment restriction** (which leads to faster data transfer)

MIPS uses big-endian ordering (use the address of the leftmost byte as word address)

Byte addressing affects array index where to get the proper byte address of a word, you need to add the offset as 4 x 8 or 32 so that load word will load `A[8]` , instead of `A[8/4] = A[2]`

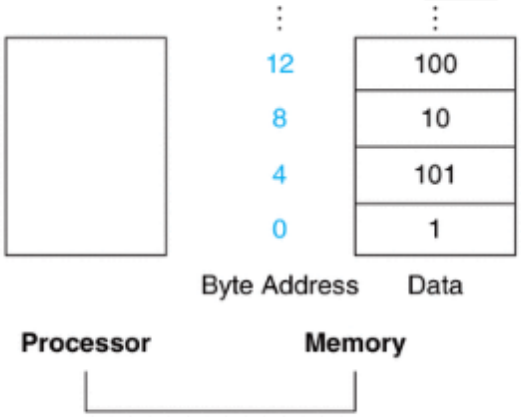
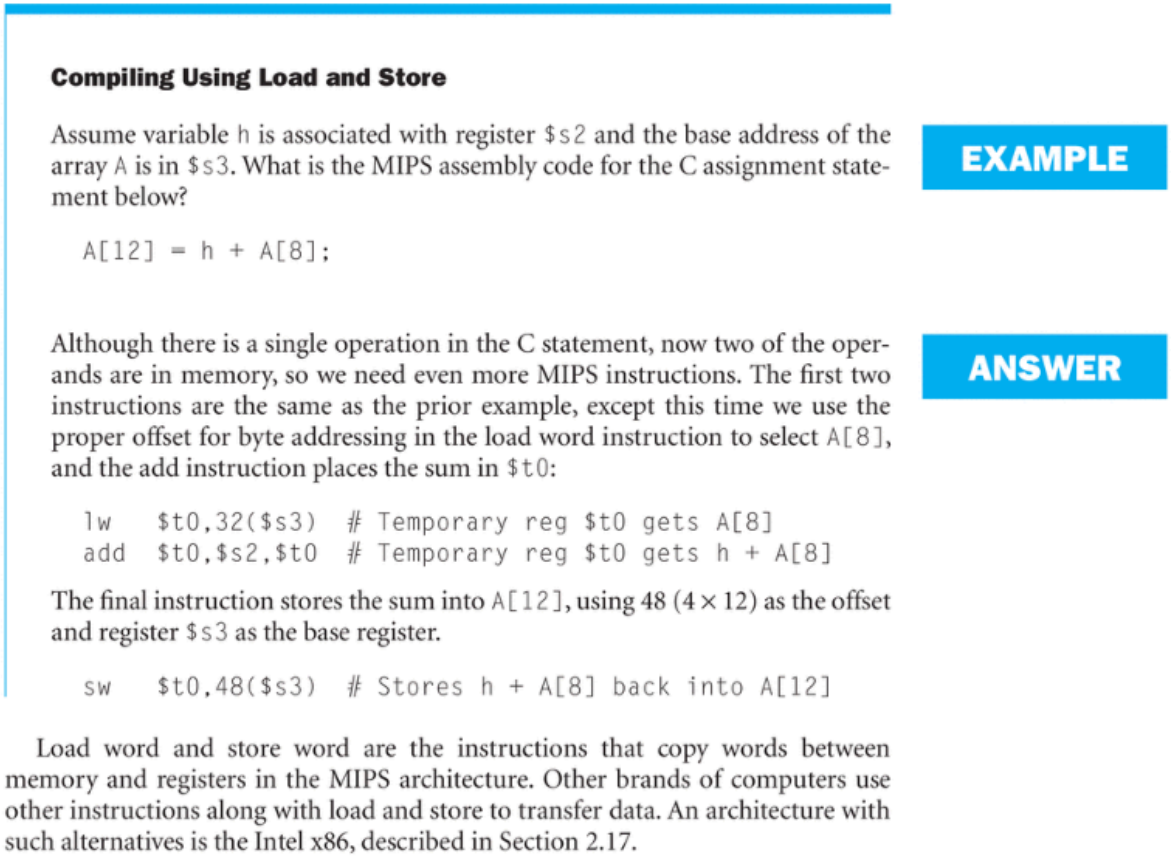



**FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words.** The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

How does storing data (words) in MIPS work?

- involves copying data from register to memory
- format of store involves name of operation, followed byt register to be stored, and then offset to select the array element and finally the base register
- MIPS address is specified in part by constant and in part by contents of register.

- TO store a word in MIPS, you use the instruction `sw` (store word)



**Compiling Using Load and Store**

EXAMPLE

Assume variable h is associated with register $s2 and the base address of the array A is in $s3. What is the MIPS assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

ANSWER

Although there is a single operation in the C statement, now two of the operands are in memory, so we need even more MIPS instructions. The first two instructions are the same as the prior example, except this time we use the proper offset for byte addressing in the load word instruction to select A[8], and the add instruction places the sum in $t0:

```
lw   $t0,32($s3)  # Temporary reg $t0 gets A[8]
add  $t0,$s2,$t0  # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into A[12], using 48 (4 × 12) as the offset and register $s3 as the base register.

```
sw   $t0,48($s3)  # Stores h + A[8] back into A[12]
```

Load word and store word are the instructions that copy words between memory and registers in the MIPS architecture. Other brands of computers use other instructions along with load and store to transfer data. An architecture with such alternatives is the Intel x86, described in Section 2.17.

  Compiler tries to put most frequently used variables in registers and rest in memory via loads and stores to move variables between registers and memory. Spilling registers is when putting less commonly used variables into memory. Data is more useful when in a register where MIPS arithmetic instructions can read two registers, perform operations on them and write the result. MIPS data transfer instruction only reads/writes one operand without operating on it

**Constant or Immediate Operands**

To perform operations with constants, you can load a constant from memory to use that constant (as constants would have been placed into memory when program was loaded)

Alternatively can perform arithmetic operations where one operand is a constant e.g. `addi` (add immediate), where to add 4 to register $s3, you perform

```
addi $s3, $s3, 4 # $s3 = $s3 + 4
```

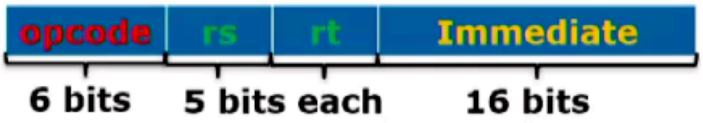
MIPS has a zero register ($zero) which is always set to 0 even if you try to overwrite that register.

MIPS offset plus base register addressing is a good way to match to structures like arrays as register can point to beginning of structure and offset can be used to select the desired

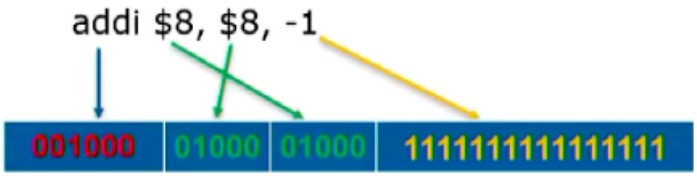
element. MIPS also supports negative constants, so you can subtract numbers by `addi` negative constants

## Immediate Instructions

- Work with a 16-bit immediate values
- when adding immediate values e.g. -1, you need to sign extend the numbers so that it is represented in 16 bits
- add immediate (and any immediate instructions) is limited to constants no larger than $\pm 2^{15}$




R[rt]=func(Imm,R[Rs]) *




| | | | | | |
|---|---|---|---|---|---|
| Add Immediate | `addi` | I | R[rt] = R[rs] + SignExtImm | (1,2) | $8_{hex}$ |

e.g. addi $8 $8 -1 = add -1 to the value in register 8 and store it back in register 8

same as $8 -= 1

when you want to know what type of instruction, look at the opcode

## Memory Addressing

As memory addresses and values stored in registers are only 32-bit long

How to compute a memory address:

- Base (register value) + Displacement (Immediate value)

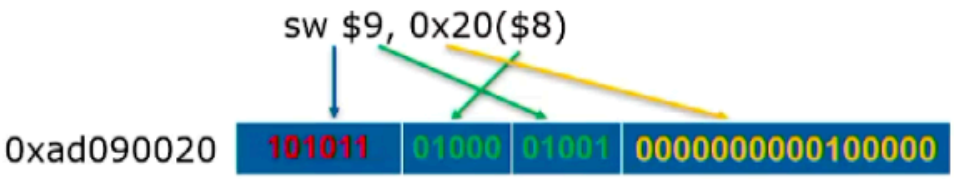
e.g. `0x20($8)`
look at the address (value stored in register 8) + `0x20`

where if the address in register 8 is `0x10010000` , then `0x20($8)` will be the address `0x10010020`

## Immediate memory instructions

- Working with integer arrays:
  - $A[8] = y;$
  - Say $8 stores the address of A and $9 stores y

| | | | | | |
|---|---|---|---|---|---|
| Store Word | sw | I | M[R[rs]+SignExtImm] = R[rt] | (2) | $2b_{hex}$ |




Base of memory address is in `rs` and the cosntant/address takes up 16 bits (e.g. 0x20), SignExtImm (sign extension of immediate value) will be the displacement

That will get you the address to write to, and we will write to address the value stored in `rt`

example: `sw $9, 0x20($8)` (store the 8th element of the array)

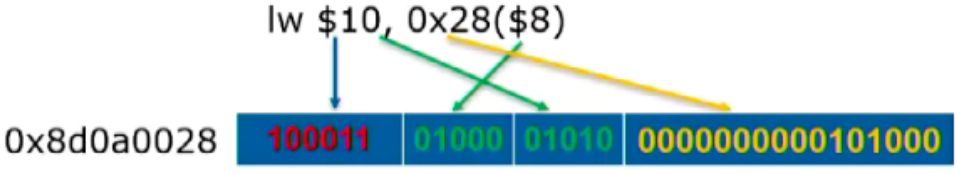
to go to location 8, have to offset by 32 bytes from beginning of array (hence writing to hex 20), as you offset by 4 bytes when moving through 1 index every time.

example 2: `z = A[10] # store A[10] into $10`

`lw $10, 0x28($8)` (load the 10th element of the array)

- $z = A[10];$ # store A[10] into $10.

| | | | | | |
|---|---|---|---|---|---|
| Load Word | lw | I | R[rt] = M[R[rs]+SignExtImm] | (2) | $23_{hex}$ |




note that 16-bit address/constant means that the load/store word instruction can load/store any word within a region of $\pm 2^{15}$ bytes (equiv to $\pm 2^{13}$ words) of the address in base register `rs`

You also can load/store other data types i.e.:

- bytes (lb/sb)
- half-word (lh/sh) - 2 bytes/16 bits
  These operations are all sign-extended

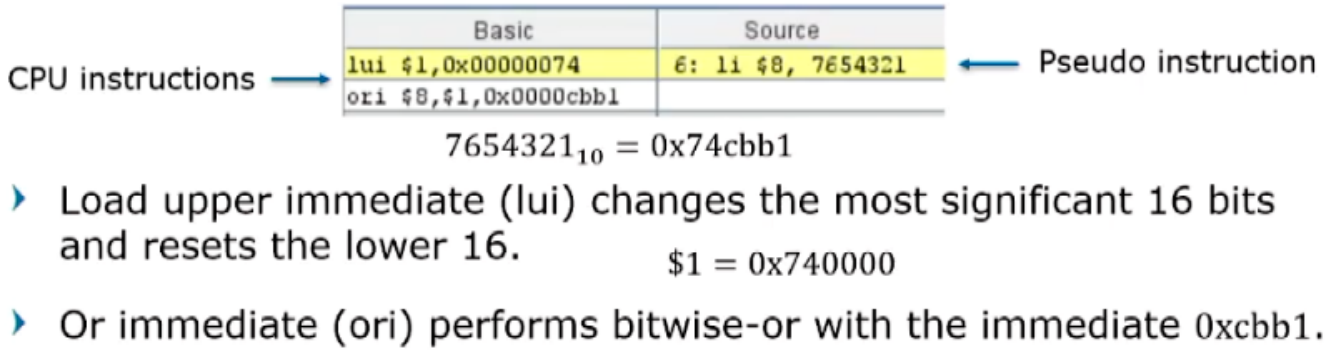
There is support for zero extensions (lbu/lhu/sbu/shu)

## Immediate Value Assignment

As we only have 16 bits to encode immediate value - need two instructions if want to assign 32-bit immediate values as there is no room to specify a full 32-bit value in a single immediate instruction

Pseudo instruction for loading an immediate value:
`li $8, 7654321 # load 7654321 into register $8`

(not a real MIPS code, this is translated as)




you are loading immediate a value that cannot be represented with 16 bits, so ahve to split them up so the value can be loaded into register (lui and ori) to be able to load a full 32-bit constant into a register

lui (load upper immediate) will write most significant 16 bits of a register and sets the least significant (lower) 16 bits to 0
ori (or immediate) will perform bitwise or between reigster and zero extended 16 bit immediate value (LSB) with the least signficant 16 bits and write result into register $8 (value is placed in least signficant 16 bits in the register)

```
lui $8, upper16bits  # Load upper 16 bits into $8, lower 16 bits are zeroed
ori $8, $8, lower16bits  # OR in the lower 16 bits
```

e.g. loading 0x12345678 ( `li $8 0x12345678` ) pseudo-instruction is translated as:

```
lui $8, 0x1234 # $8 = 0x12340000
ori $8, $8, 0x5678 # $8 = 0x12345678
```

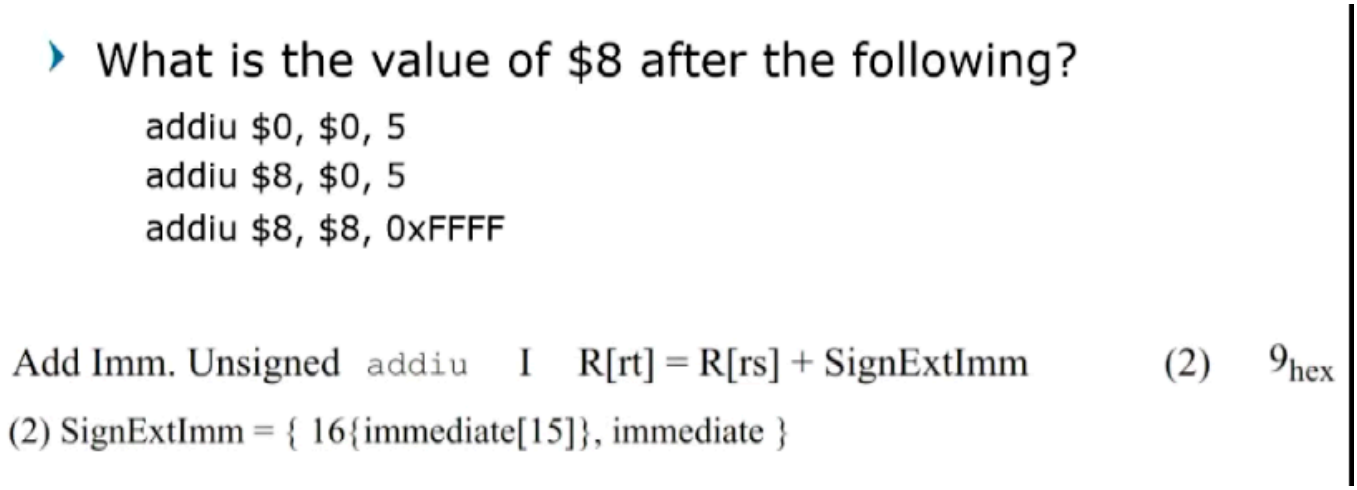
register $1 is assembly temporary (register that assembly that may modify to implement pseudo instruction)

a pseudo instruction can be translated to at least 1 CPU(MIPS) instruction

**Example Question**

- What is the value of $8 after the following?

```
addiu $0, $0, 5
addiu $8, $0, 5
addiu $8, $8, 0xFFFF
```




answer = 4

`addiu $0 $0 5` will not add 5 to zero register as $0 will always be 0
`addiu $8 $0 5` will set the value 5 to register 8 (as you add 0 + 5)
`addiu $8 $8 0xFFFF` will add -1 (0xFFFF - as its signed) to what is stored in register 8 (5), which will result in 4

based on reference card, you set $8 be $8 plus the sign extension of the immediate (take MSB of immediate and add 16 bits with the same value before the immediate)

`addiu` is used to add constants to **signed integers** when we don't care about overflow. Negative numbers need sign extension as MIPS has no subtract immediate instruction, so the immediate field gets sign extended - mips architects called it addiu as unsigned means it doesnt trigger exception (but it adds signed numbers - **important**)

addiu will never throw an overflow ( `addi` - add immediate can throw overflow) so if you wantt o decrement numbers, you use `addiu` instead of `addi` if you do not care about overflow exception

why is 65540 also correct?

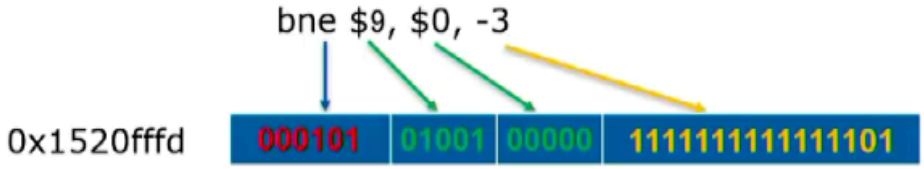
- MARS assembler treats `addiu $8, $8, 0xFFFF` as a pesudo instruction where it replaced that command with 3 instructions that add 0xFFFF = $65535_{10}$ to $8
- but if you add -1, it will addiu $8, $8, 0xffffffff (where it represents -1) - assembler can treat it as pseudo instruction
- you have to see mars manual to understand how pseudoinstructions are translated (like which instructions is considered as pseudo instructions in MARS)

| Basic | Source |
|---|---|
| addiu $0,$0,0x00000005 | 5: addiu $0, $0, 5 |
| addiu $8,$0,0x00000005 | 6: addiu $8, $0, 5 |
| lui $1,0x00000000 | 7: addiu $8, $8, 0xFFFF |
| ori $1,$1,0x0000ffff | |
| addu $8,$8,$1 | |

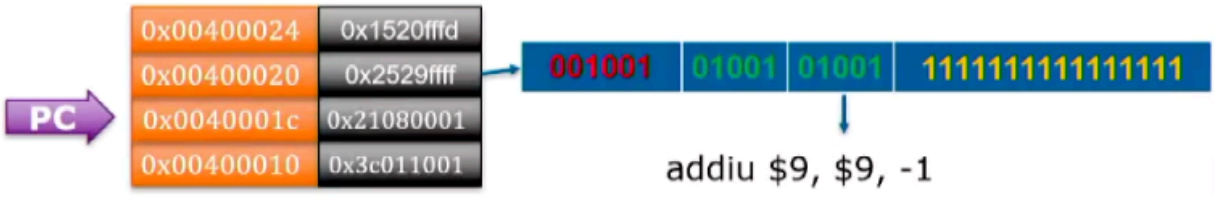
| Code | Basic | Source |
|---|---|---|
| 0x2508ffff | addiu $8,$8,0xffffffff | 2: addiu $8, $8, -1 |

## Immediate branching instructions

- Used for implementing loops
- knwon as **conditional branches**
- equivalent to an if statement (decision making) or while loop (iteration)
- two instructions: `beq` (branch if equal), `bne` (branch if not equal)
- e.g. `bne $9, $0, -3`




Branch On Not Equal bne I if(R[rs]!=R[rt]) PC=PC+4+BranchAddr (4) $5_{hex}$




Checks if value in register rs is not equal to value in register rt. If theyre not equal itll modify the value of program couinter, otherwise program counter will increment by 4

BranchAddr = branch address (in the number of instructions) e.g. -3 means go 3 instructions back (go back 12 bytes but branch address is defined in number of instructions)

as instructions take 32 bits (4 bytes), instructions are only stored in addresses that are in multiples of 4 - no point in encoding least two signfincaint bitgs (more range for jump for branch address), thus the branch address always have 2 0s in the last two bits.

(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }

Example above is a loop that executes (iteratively decrements by 1 until register 9 equals to 0, as if theyre not equal, you keep going back 3 instructions)

```
bne $9, $0, -3 # check if values are not equal, otherwise go back 3
instructions
addiu $9, $9, -1
```

When executing a branch instruction like `beq` (branch if equal) in MIPS, you add the branch address because MIPS implements relative addressing for branches rather than absolute addressing.

Here's why:

1. **PC-relative addressing**: The branch target is specified as an offset relative to the current Program Counter (PC). This requires fewer bits than specifying an absolute memory address.
2. **Implementation details**: When executing a branch instruction, the following happens:
   - The 16-bit immediate in the instruction is sign-extended to 32 bits
   - This value is shifted left by 2 bits (multiplied by 4) because instructions are word-aligned (4 bytes)
   - This offset is then added to PC+4 (the address of the instruction following the branch)
3. **Advantages**:
   - Allows code to be position-independent (can be loaded anywhere in memory)
   - Requires only 16 bits for the offset field instead of 32 bits for an absolute address
   - Most branches tend to be to nearby locations, so a relative offset is sufficient

The instruction format ensures that the branch can reach addresses within ±32KB of the current instruction, which is adequate for most control structures like loops and conditionals.

**unconditional branch** - jump instructions (abbreviated as `j`)

you can use/define labels to indicate the branches if want to compile if-then-else statements into conditional branches

how to translate if x<= y in MIPS?

- Pseudo instruction: branch greater than (bgt)
- bge, ble, blt

| | | |
|---|---|---|
| 0x016a082a | slt $1,$11,$10 | 13: bgt $10, $11, loop |
| 0x1420fffb | bne $1,$0,0xfffffffb | |

```
for (int i=0; i<10; ++i){
    func();
}
```

| Basic | Source |
|---|---|
| addiu $13,$0,0x00000000 | 7: li $13, 0 |
| addiu $14,$0,0x0000000a | 8: li $14, 10 |
| jal 0x00400024 | 10: jal func |
| addi $13,$13,0x00000001 | 11: addi, $13, $13, 1 |
| slt $1,$13,$14 | 12: blt $13, $14, loop |
| bne $1,$0,0xfffffffc | |

```
for (int i=10; i>0; --i){
    func();
}
```

| Basic | Source |
|---|---|
| addiu $13,$0,0x0000000a | 8: li $13, 10 |
| jal 0x00400024 | 10: jal func |
| addi $13,$13,0xffffffff | 11: addi, $13, $13, -1 |
| bgtz $13,0xfffffffd | 12: bgtz $13, loop |

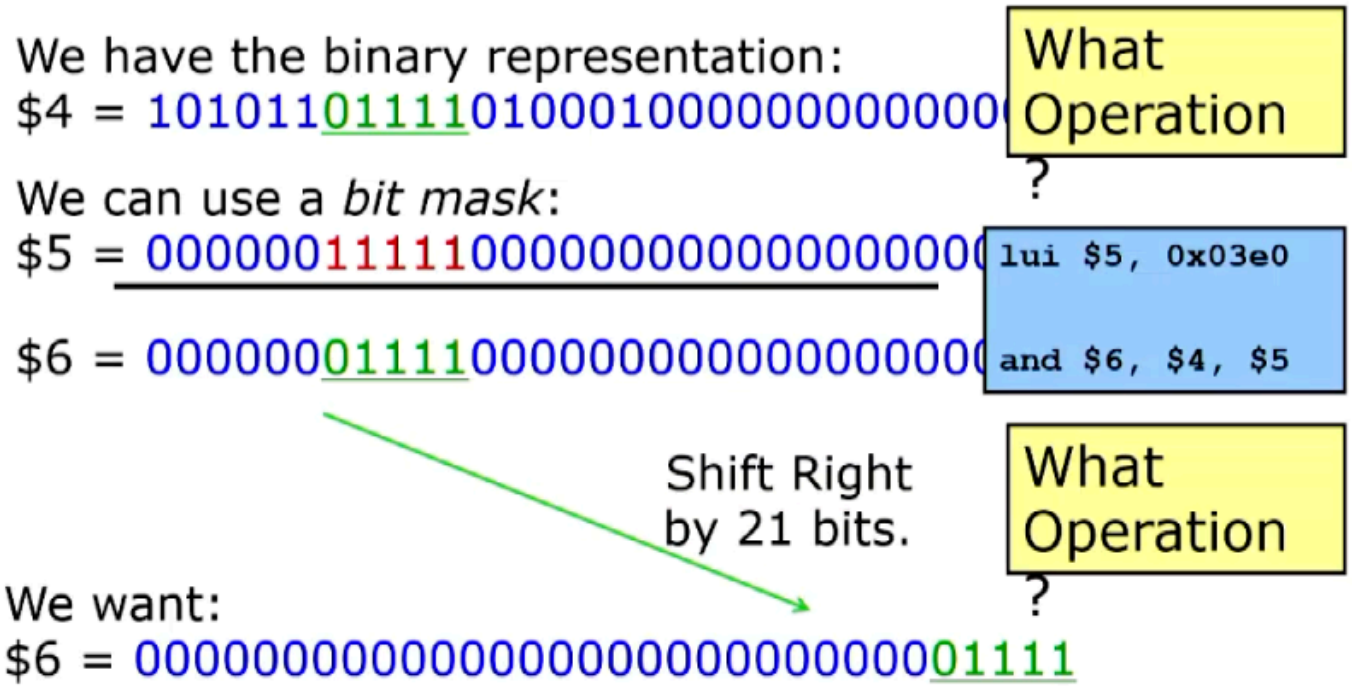
what differs in compilers is how they optimise higher level code into assembly language so the translation is more efficient in assembly

# Combining MIPS Instructions

How do we translate unique machine code to MIPS code via assembly?

- Bit manipulation via bit mask (load register via `lui` to load the bitmask) and perform bitwise operations to get register values

example, triyng to get the rs value from machine code




- Load register 5 with bitmask (correspond to bits in `rs`)
- perform bitwise and to get the bits needed for rs (set everything else to 0)
- perform shift right by 21 to bring the bits to the right (LSB)
- alternatively can perform left shift to get rid of opcode bits (left by 6 bits) and then right shift 27 bits (2 instructions compared to 3 = more efficient)

**Comparing registers with fixed numbers**

- Can use "set comparison" instructions:

```
slti rt, rs, value          # Set less than (signed)
```

**If (rs < value) then set rt = 1 else set rt = 0**

```
sltiu rt, rs, value         # Set less than (unsigned)
```

**If (rs < value) then set rt = 1 else set rt = 0**

- can perform beq/bne against zero register to test result register rt (store the result of the set comparison)

**Computing average with MIPS**

```
# Calculating average (Version 1)

      li $8, 3501006752
      li $9, 794337423
      addu $10, $8, $9
      srl $10, $10, 1

      li $v0, 36 #syscall 36 -- print unsigned int
      move $a0, $10 # a0 = $10
      syscall #print $10
```

doesn't work as `addu` throws an overflow exception as adding the values from both register 8 and register 9 reuslts in a sum value that is larger than 32-bits

```
# Calculating average (Version 2)

      li $8, 3501006752
      li $9, 794337423
      and $10, $8, $9
      xor $11, $8, $9
      srl $11, $11, 1

      add $10, $10, $11

      li $v0, 36 #syscall 36 -- print unsigned int
      move $a0, $10 # a0 = $10
      syscall #print $10
```

as you cannot add both numbers up, you perform bitwise and, adn the bitwise xor

- We want to compute the average of two registers

Cannot add these up!

```
$8  = 11010000101011010001111110100000
$9  = 00101111010110001010000010001111
$10 = 00000000000010000000000010000000   and $10, $8, $9

$8  = 11010000101011010001111110100000
$9  = 00101111010110001010000010001111
$11 = 11111111111101011011111100101111   xor $11, $8, $9
```

Mathematical bitwise identity:

$$x + y = 2 * (x\&y) + x\hat{}y$$

Therefore:

$$\frac{x+y}{2} = (x\&y) + \frac{x\hat{}y}{2}$$

```
srl $11, $11, 1
add $10, $10, $11
```

every bit in bitwise and correspond to both bits being one (contributes twice its value to the sum)

## Exam Questions

Q1a. Which operation cannot be implemented using a native single instruction, assuming that $t1 represents unsigned integer?

[(-2,+5) marks]

A) set $t0 to ($t1 modulo $2^{15}$)
B) set $t0 to ($t1 $-2^{15}$)
C) set $t0 to ($t1 $+2^{15}$)
D) set $t0 to 1 if ($t1 is smaller than $2^{32}-2^{15}$) and to 0 otherwise

a. false - modulo involves performing bitwise and on the value $2^{15}-1$ immediate and you can store the value in
$t0b. false - youcanperformthatwithoneoperation(\addiu\$t0, \$t1, 0xFFFFFF\)where-2^15issignexte$
with 16 bits (max that can be encoded with 16 bits is $2^{15}-1$). if you try that, MARS will consider that as a pseudoinstruction (if you try addiu reigster 0 with 2^15)**
d. false - you can use `sltiu` where it sign expands the immediate valu4e (comparison is done as if values are unsigned)

this question tests about what numbers (signed) that can be represented by 16 bits (i.e. $[-2^{15}, 2^{15}-1]$)

Q1b. A `bne` instruction is placed at address $I$. After executing it, what's the smallest potential address of the next instruction?

[(-2,+5) marks]

A) $I-2^{16}$
B) $I-2^{16}+4$
C) $I-2^{17}$
D) $I-2^{17}+4$

note that the smallest negative number that can be represented by 16 bit is $-2^{15}$ instructions, which is equivalent to $-2^{15}\times 4$ addresses = $-2^{17}$

the address is relative to program counter (you increment by 4 first before adding the branch address, i.e. $-2^{17}$ from I , giving you the answer $I-2^{17}+4$)

answer = D

# Week 4 - High Level Languages

Ways to access registers where different registers have a different name and purpose

| Name | Register Number | Usage |
|---|---|---|
| **$zero** | **0** | **the constant value 0** |
| $at | 1 | assembler temporary |
| $v0-$v1 | 2-3 | function return values |
| $a0-$a3 | 4-7 | function arguments |
| **$t0-$t7** | **8-15** | **temporaries** |
| **$s0-$s7** | **16-23** | **saved variables** |
| **$t8-$t9** | **24-25** | **more temporaries** |
| $k0-$k1 | 26-27 | OS temporaries |
| $gp | 28 | global pointer |
| $sp | 29 | stack pointer |
| $fp | 30 | frame pointer |
| $ra | 31 | function return address |

E.g. for register zero, you can pass either $0 or \$zero as your argument as both refers to the zero register. \$1 is the assembler temporary (also can be writte
t0 is the $8 (register 8), and so on - refer to the table.

## Handling arrays in MIPS

You store the address of the array location in memory into a register in MIPS

**High-level Code**

```
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

**MIPS assembly code**

```
# array base address = $s0 (say array stored at 0x12348000)
  lui  $s0, 0x1234        # put 0x1234 in upper half of $s0
  ori  $s0, $s0, 0x8000   # put 0x8000 in lower half of $s0

  lw   $t1, 0($s0)        # $t1 = array[0]
  sll  $t1, $t1, 1        # $t1 = $t1 * 2
  sw   $t1, 0($s0)        # array[0] = $t1

  lw   $t1, 4($s0)        # $t1 = array[1]
  sll  $t1, $t1, 1        # $t1 = $t1 * 2
  sw   $t1, 4($s0)        # array[1] = $t1
```

`0($s0)` will load the first element of array, `4($s0)`, loads the second element of array, and so on. **offsets are in multiples of 4** as words are 32 bit long

# Jump Instructions


6 bits reserved for opcode, 26 bits reserved for address.

However memory addresses are 32 bits long

- you onloy can jump to a limit of 26 bit addresses ($2^{26}$ possible addresses)
- hence calculation is performed by processor to determine the new jump address (which address will the instruction jump to)

**How is the jump address calculated from the `address` argument**


| Address | Code | Basic |
|---|---|---|
| 0x00400000 | 0x2408000a | addiu $8,$0,0x0000000a |
| 0x00400004 | 0x08100004 | j 0x00400010 |
| 0x00400008 | 0x21080001 | addi $8,$8,0x00000001 |
| 0x0040000c | 0x21080001 | addi $8,$8,0x00000001 |

Jump j J PC=JumpAddr (5) $2_{hex}$

(5) JumpAddr = { PC+4[31:28], address, 2'b0 }

based on reference card (5),

This equation represents how a jump address is calculated in MIPS processor architecture. Let me break it down:

```
JumpAddr = { PC+4[31:28], address, 2'b0 }
```

This is using a concatenation operation (represented by the curly braces `{}`) to form a 32-bit jump address from three parts:

1. `PC+4[31:28]`: These are the 4 most significant bits (bits 31-28) of the Program Counter (PC) after it has been incremented by 4. This keeps the jump within the same 256MB region of memory.
2. `address`: This is typically a 26-bit address field from the jump instruction.
3. `2'b0`: These are 2 bits of value 0 appended at the end. This effectively multiplies the address by 4, ensuring that the jump target is aligned to a word boundary (since MIPS instructions are 4 bytes).

The reason for this format is that MIPS jump instructions (J-type) only have space for a 26-bit address in the instruction format. To create a full 32-bit address, the processor takes the upper 4 bits from the current PC and appends two zero bits at the end for word alignment.

This mechanism allows MIPS to have a relatively compact instruction format while still enabling jumps to a large address space.

there is a relation between the address passed in jump instruction and what is the actual jump address that will be jumped to after calculations

first, the processor will **take the jump address and append 2 bits to the right (LSB)** - multiply by 4

- this is because the first two bits (rightmost / LSB) of jump address have to be 0, as addresses need to be word-aligned (divisible by 4)

**the 26 bits in jump instruction arguemnt is copied to next 26 bits of the address to jump to**

**the most significant 4 bits of PC are copied on jump instructions**, where the actual jump address has its last 4 bits (MSB / leftmost) that come from incrementing 4 by the value in Program Counter.

we can't jump to any address where the 4 most significant bits is different to the 4 most signficant bits of value stored in PC + 4

based on reference card, concatenating 2 0s means multiplying the jump address by 4 (shift by 2 bits) - argument (you only need to specify the instruction word address, not the byte address), as the processor calculates the target address using the formula, where it gets loaded to program counter so at next clock cycle, processor can fetch the instruction from the new PC value.

$$\text{NewPC} = (\text{PC}+4)[31:28] \cdot JA \cdot 00 = 4 \cdot JA = 0\text{x}400010$$

$$\Rightarrow JA = \frac{0\text{x}400010}{4} = 0\text{x}100004$$

after performing jump instruction, value in PC (program counter) will be updated

you can reverse engineer the new value stored in PC to calculate what is the jump address argument specified in the previous jump instruction that was executed before the value in PC was updated.

As 4 most signifcant bits of PC gets copied to jump instructions where its same as the 4 most significant bits of the actual jump address (limiting the range of addresses that we can jump to)

**Limitations of jump instruction:**

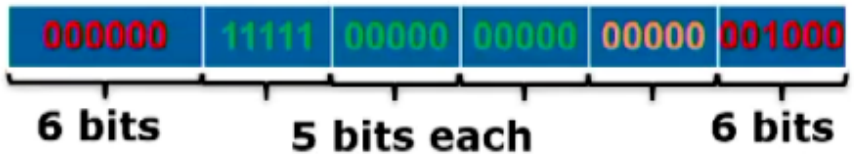
- when branching into a code region with different MSB
- when branching into a code region that is only known at runtime

Solution:

- jump register `jr` - set PC to the value (i.e. address) stored in the register specified in the register argument

– An *R*-type instruction!

– jr $ra #Remember: $ra is the *name* of $31




0x03e00008

(opcode 0 but has function number $8_{16}$)
basically jump to the address stored in register 32

## If .. else .. branching

We can use jump instructions and immediate branching instructions ( `bne` / `beq` )
**Tip**: if the **C condition is `!=`, use `beq` - branch if equal. if `==`, use `bne` - branch if not equal**. so that it branches if the condition is not fulfilled (basis of condition/loops). it is more efficient to **test for opposite condition to branch** over code that performs the subsequent `then` part of the `if`

**High-level code**

```
if (i == j)
  f = g + h;
else
  f = f - i;
```

**MIPS assembly code**

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j

      bne $s3, $s4, L1
      addu $s0, $s1, $s2
      j   done
L1:   subu $s0, $s0, $s3
```

bne - is the condition (if condition) - you compare i and j
if there is no jump to label L1, it means that i == j (if statement is fulfilled) so `addu` g and h and store in f ($s0) and jump to the `done` label

otherwise (case when i != j), it gets branched to the L1 label where the statement `f = f - 1` is performed

**textbook example:**

### Compiling *if-then-else* into Conditional Branches

In the following code segment, `f`, `g`, `h`, `i`, and `j` are variables. If the five variables `f` through `j` correspond to the five registers `$s0` through `$s4`, what is the compiled MIPS code for this C *if* statement?

```
if (i == j) f = g + h; else f = g - h;
```

Figure 2.9 is a flowchart of what the MIPS code should do. The first expression compares for equality, so it would seem that we would want the branch if registers are equal instruction (`beq`). In general, the code will be more efficient if we test for the opposite condition to branch over the code that performs the subsequent *then* part of the *if* (the label `Else` is defined below) and so we use the branch if registers are *not* equal instruction (`bne`):

```
bne $s3,$s4,Else   # go to Else if i ≠ j
```

The next assignment statement performs a single operation, and if all the operands are allocated to registers, it is just one instruction:

```
add $s0,$s1,$s2    # f = g + h (skipped if i ≠ j)
```

We now need to go to the end of the *if* statement. This example introduces another kind of branch, often called an *unconditional branch*. This instruction says that the processor always follows the branch. To distinguish between conditional and unconditional branches, the MIPS name for this type of instruction is *jump*, abbreviated as `j` (the label `Exit` is defined below).

```
j Exit    # go to Exit
```

The assignment statement in the *else* portion of the *if* statement can again be compiled into a single instruction. We just need to append the label `Else` to this instruction. We also show the label `Exit` that is after this instruction, showing the end of the *if-then-else* compiled code:

```
Else:sub $s0,$s1,$s2   # f = g - h (skipped if i = j)
Exit:
```

## While loop branching

## High-level code

```
// determines the power
// of x such that 2^x = 128

int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

## MIPS assembly code

```
# $s0 = pow, $s1 = x

        addi $s0, $0, 1
        add  $s1, $0, $0
        addi $t0, $0, 128
while:  beq  $s0, $t0, done
        sll  $s0, $s0, 1
        addi $s1, $s1, 1
        j    while
```

Branches are I-type instructions (16 immediate bits), as compared to J-type instructions (26 bits), so can jump to a greater range than branching, hence jumps might be needed when performing branching.

**textbook example:**

**Compiling a *while* Loop in C**

Here is a traditional loop in C:

```
while (save[i] == k)
       i += 1;
```

Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array `save` is in `$s6`. What is the MIPS assembly code corresponding to this C segment?

**answer:**

The first step is to load `save[i]` into a temporary register. Before we can load `save[i]` into a temporary register, we need to have its address. Before we can add `i` to the base of array `save` to form the address, we must multiply the index `i` by 4 due to the byte addressing problem. Fortunately, we can use shift left logical, since shifting left by 2 bits multiplies by $2^2$ or 4 (see page 103 in the prior section). We need to add the label `Loop` to it so that we can branch back to that instruction at the end of the loop:

```
Loop: sll  $t1,$s3,2    # Temp reg $t1 = i * 4
```

To get the address of `save[i]`, we need to add `$t1` and the base of `save` in `$s6`:

```
      add $t1,$t1,$s6     # $t1 = address of save[i]
```

Now we can use that address to load `save[i]` into a temporary register:

```
      lw  $t0,0($t1)      # Temp reg $t0 = save[i]
```

The next instruction performs the loop test, exiting if `save[i]` ≠ `k`:

```
      bne  $t0,$s5, Exit  # go to Exit if save[i] ≠ k
```

The next instruction adds `1` to `i`:

```
      addi  $s3,$s3,1     # i = i + 1
```

The end of the loop branches back to the *while* test at the top of the loop. We just add the `Exit` label after it, and we're done:

```
      j     Loop          # go to Loop
Exit:
```

(See the exercises for an optimization of this sequence.)

## For loop

**High-level code**

```
// add the numbers
// from 0 to 9

int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

**MIPS assembly code**

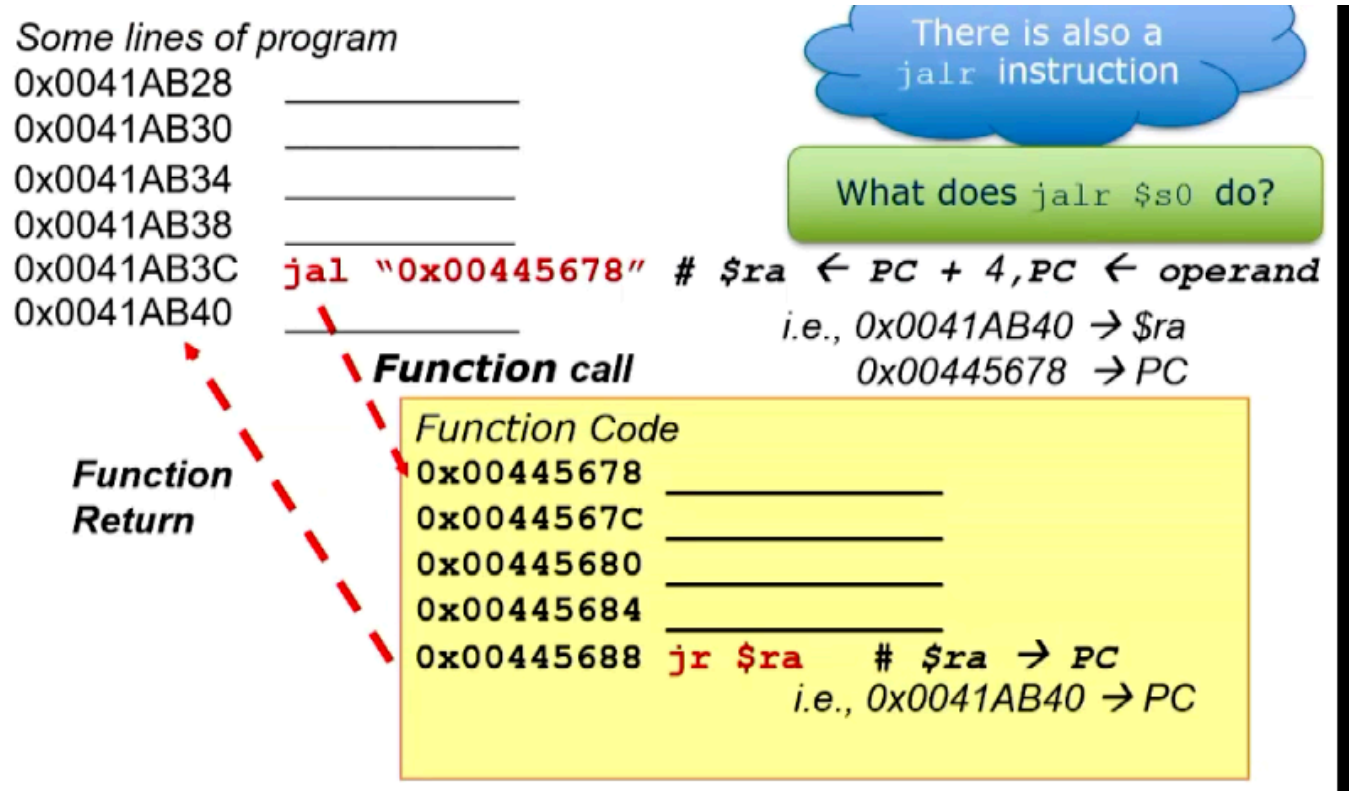
```
# $s0 = i, $s1 = sum

        addi $s1, $0, 0
        add  $s0, $0, $0
        addi $t0, $0, 10
for:    beq  $s0, $t0, done
        add  $s1, $s1, $s0
        addi $s0, $s0, 1
        j    for
done:
```

## Writing functions in MIPS

- Involves the stack segment in MIPS and register usage conventions
- how can functions know not to overwrite each other's registers
- you need to know how a processor calls a function in MIPS

WHy functions?

- they are abstractions in higher order languages



**Jump Instructions for Function Calls**

1. `jal <function_label>` = **jump and link** - used for calling functions
   - Used when making a function call from anywhere in memory
   - changes PC value to the address of the first instruction in the function body (function call)
   - Modifies the value of return address ($31) to **PC + 4** so it points to the next instruction after function call
   - performs both jump and link instruction (jump to first address of function code / links the next instruction in `$ra`)
2. `jr $ra` = **jump register** - used to return from functions
   - Used when return to caller at the end of the function - same as `return`
   - sets the PC value back to the next instruction after function call, where the value of next instruction was stored in `$ra` after a `jal` call
   - Ensures program always know where to return to after calling function - which instruction to execute after return

However, `jal` only can jump to 26 bit addresses ($2^{26}$ addresses), hence there is a `jalr` (jump and link register)

Funcitons will act (can alter values) on the same 32 registers as the main code (the caller) - original code has to know what happened to the values in register after function call

**Saved registers for local variables**

**High-level code**

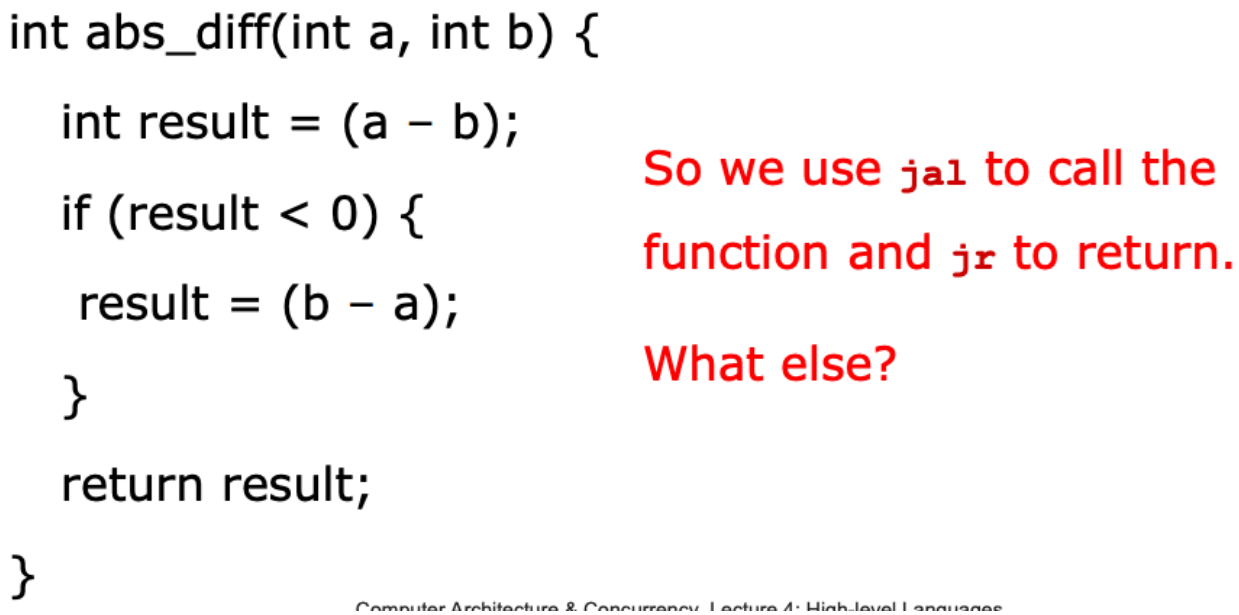
```
// Integers represented
// as 32-bit signed words.

int a = 64;


int b = 305419896;


func()


int c = a + b;
```

**MIPS assembly code**

```
# $s0 = a
addi $s0, $0, 0x0040

# $s1 = b
lui $s1, 0x1234
ori $s1, $s1, 0x5678

# Or pseudo-instruction
# li $s1, 0x12345678

jal func

# $s2 = c
add $s2, $s0, $s1
```

Convention in MIPS - stored registers (start with s) are going to be stored in function calls - function shouldnt change values in stored registers\

example of function:




consider about where to:

- store input variables for function
  - in argument registers ($a0 - $a4) - can pass up to 4 arguments to function
- store result variable within function
  - in register $v0 (as well as $v1 if result value is 64-bits long i.e. double/long long type)
- put output for caller
  -

## Registers and usage conventions

| Register Name | Register Number | Usage |
|---|---|---|
| $v0 | $2 | Result value |
| $v1 | $3 | Result value (if 64-bit result or two 32-bit results are generated) |
| $a0 | $4 | Argument 1. |
| $a1 | $5 | Argument 2. |
| $a2 | $6 | Argument 3. |
| $a3 | $7 | Argument 4. |
| $ra | $31 | Return address. |

| Register Name | Register Number | Usage |
|---|---|---|
| $t0, $t1, $t2, $t3, $t4, $t5, $t6, $t7 | $8 to $15 | Temporary registers. *Do not need to be preserved* across function calls |
| $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7 | $16 to $23 | Saved registers. The function *must* save and restore these if it wants to use them. |
| $t8, $t9 | $24, $25 | More temporary registers. |

shows the individual roles of registers. you use both registers v0 and v1 if you want to output a `double` or `long long` value as they both are 64-bits long

what is preserved/ not preserved in function calls

| Preserved | Not preserved |
|---|---|
| Saved registers: $s0–$s7 | Temporary registers: $t0–$t9 |
| Stack pointer register: $sp | Argument registers: $a0–$a3 |
| Return address register: $ra | Return value registers: $v0–$v1 |
| Stack above the stack pointer | Stack below the stack pointer |

**FIGURE 2.11 What is and what is not preserved across a procedure call.** If the software relies on the frame pointer register or on the global pointer register, discussed in the following subsections, they are also preserved.

**argument registers**

- used to store arguments to pass to function calls (parameters)

**temporary registers**

- registers that can be used which can be overwritten by functions

**saved registers**

- registers that caller function won't change values of
- you will know the values will still be there unchanged if stored in saved registers after function call/return

Similarly, in the execution of a procedure, the program must follow these six steps:

1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure.
3. Acquire the storage resources needed for the procedure.
4. Perform the desired task.
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin, since a procedure can be called from several points in a program.

As mentioned above, registers are the fastest place to hold data in a computer, so we want to use them as much as possible. MIPS software follows the following convention for procedure calling in allocating its 32 registers:

- `$a0–$a3`: four argument registers in which to pass parameters
- `$v0–$v1`: two value registers in which to return values
- `$ra`: one return address register to return to the point of origin

## Example of Functions in MIPS

```
# abs_diff(a, b) calculates the
# absolute difference of two integers.

abs_diff:
      sub $v0, $a0, $a1   # result = a - b
      bgez $v0, return    # If +ve jump next instruction.
      sub $v0, $a1, $a0   # result = b - a

return:
      jr $ra              # Return from function.
```

more efficient than using temporary registers to store results, you first store result of a-b directly into register v0, and overwrite that register with b-a if the result of a-b is negative.
after sub is performed, itll go straight to jump register (return from function) - hpapens for both cases, but `bgez` means that itll return from function straight after determining if a-b is positive

another example:

```
# Calculating absolute difference
        .data
vals:   .word 180, 100, 0

        .text
        .globl main

main:   jal my_program    # Run my program.
        li $v0 10         # Exit program using system call 10.
        syscall           # syscall 10 (exit)

my_program:
        la $t1, vals      # Load address of data into $t1.
        lw $a0, 0($t1)    # Load A from memory into $a0.
        lw $a1, 4($t1)    # Load B from memory into $a1.

        jal abs_diff      # Jump and link to 'abs_diff' function.

        sw $v0, 8($t1)    # Write result and return.
        jr $ra            # Return to main program.
```

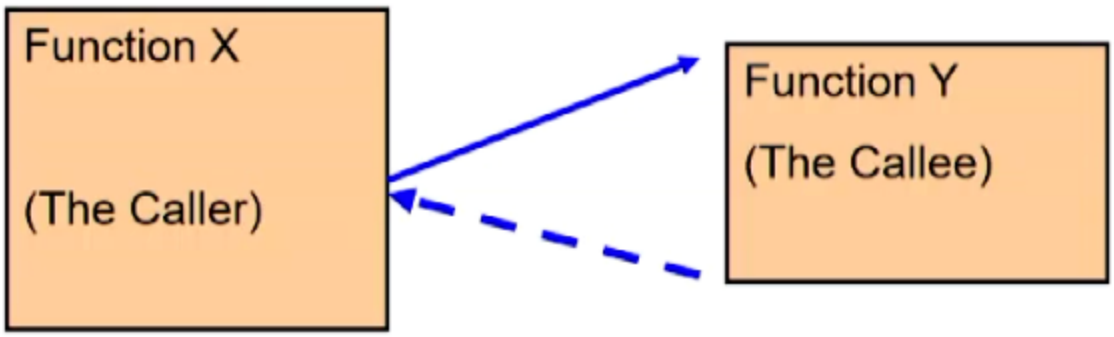
first argument is 180 (first word), second argument is 100 (second word). result of absolute difference is stored in register v0 - tells the caller of the function where the result is stored in, where can directly use that register to write the result in 3rd word.

the function modifies the value of register v0 to the result of absolute difference of a and b - follows the convention.

BUT - there is a problem of the code, as there is a chance the function `abs_diff` could overwrite other registers i.e. functions used in functin `my_program`

## Nested Function Calls

We must ensure functions do not interfere with each other's functions

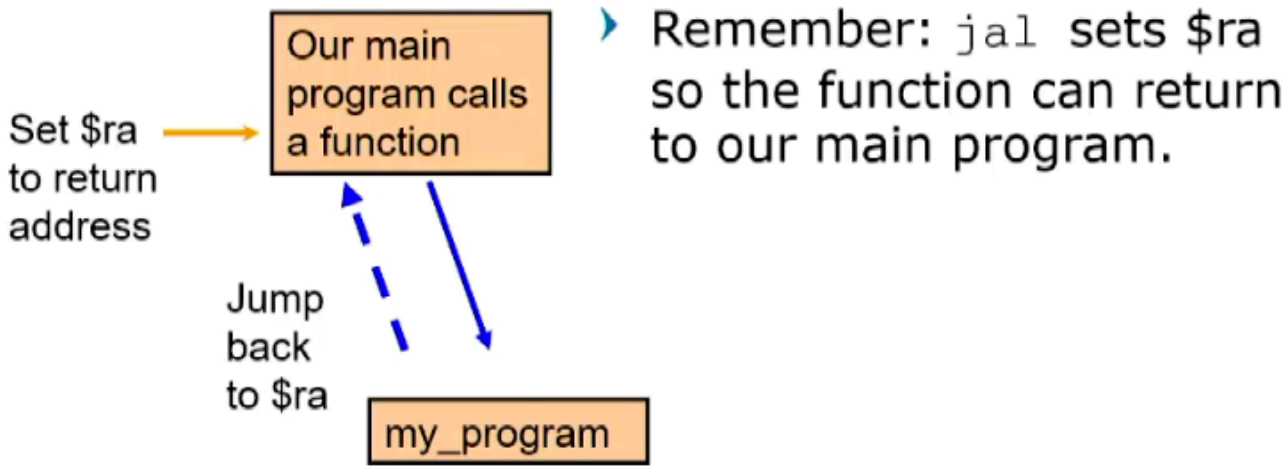



Function X calls function Y

**How do we ensure a register is not modified**

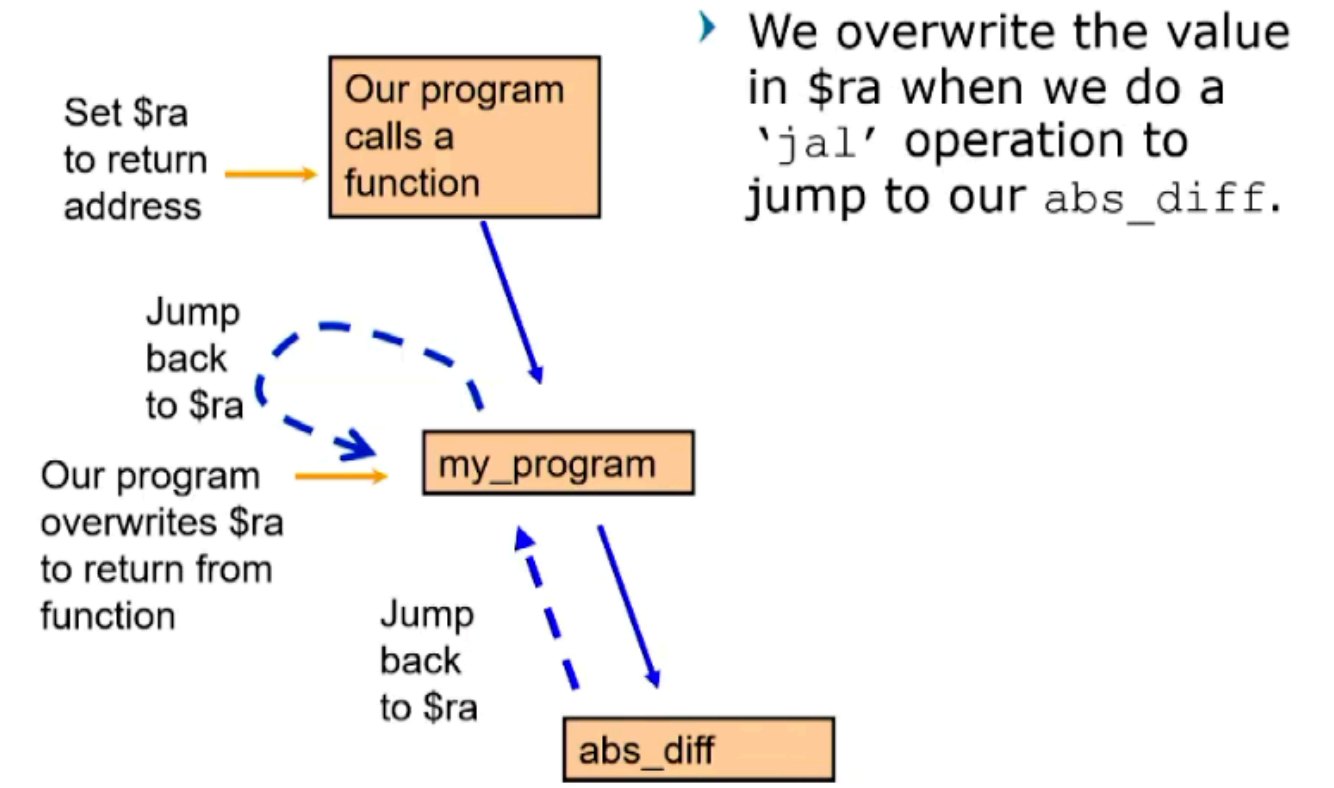
- caller needs to save register value before amking function call
- or callee needs to save register before it modifies the register value and restore the oriignal register value before returning

What happens when a function aclls another function?




Note htere is no nested function call here

BUT, what happens when itnroducing another function call ( `abs_diff` )




there is an infinite loop when trying to `jr $ra` at the end fo the `my_program` function as `$ra` points to the address of the first instruction in `my_program` (basically $ra stores the address of the function body of `my_program` and not the original caller).

this is because making a nested function call i.e. my_program calling abs_diff modified the value stored in register `$ra` such that it doesn't store the address of the next instruction of original caller, but the next instruction of `my_program` which the function `abs_diff` uses when executing `jr $ra` when returning the result of `abs_diff` (at the end of the function) - **problems of nested function calls**

How do we overcome the problem of nested function calls?

- Prolbem is we can't save the existing values of the registers in other registers as they could be in use elsewhere (including temporary/saved registers as well)
- so we **must save the values to memory** - but cannot save into same mmory location
  - because nested calling of functions will overwrite the saved value in that memory with another value
- we use a **stack** (stack memory region)!

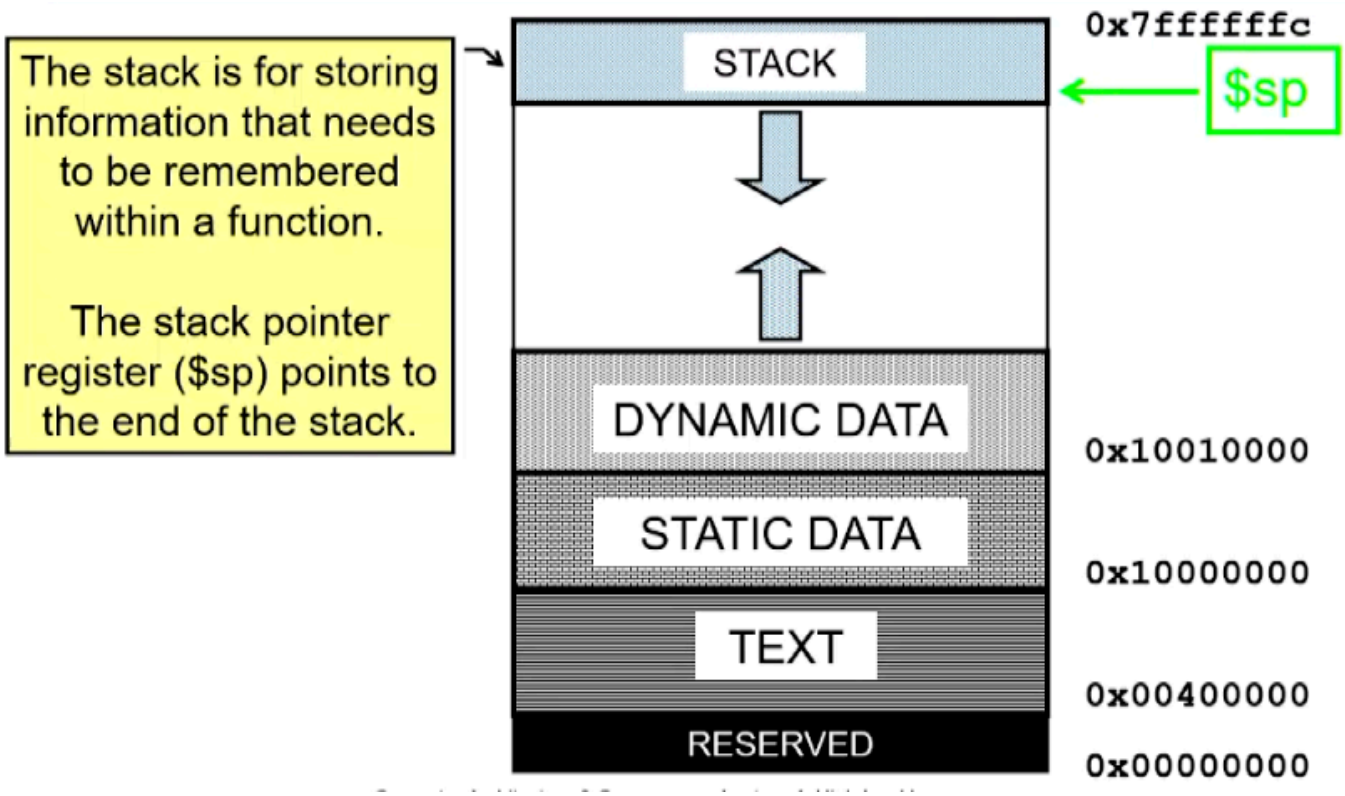
| Register Name | Register Number | Usage |
|---|---|---|
| $at | $1 | Reserved for the assembler for implementing pseudo instructions. |
| $k0, $k1 | $26, $27 | Reserved for operating system kernel |
| $gp | $28 | Useful pointer into the global static data segment (set to 0x10008000). |
| $sp | $29 | Stack Pointer – more about this soon |
| $fp | $30 | Frame Pointer – more about this soon |
| $ra | $31 | Return address from function |

cannot use $k0/\$k1 as they might be changed$gp - used to store global varialbes (static data segment)
$sp - stack pointer (stack is used to allow function calls to have memory region - so the calls won't override ea$
fp - frame pointer (used to implement the stack)
$ra - return address from function
See the MIPS Card: $gp, $sp, $fp, and $ra also need to be saved and restored if changed by a function.

## Stack Memory

Video about call stack that can help understand: https://www.youtube.com/watch?v=y9Wv1RVbbNA

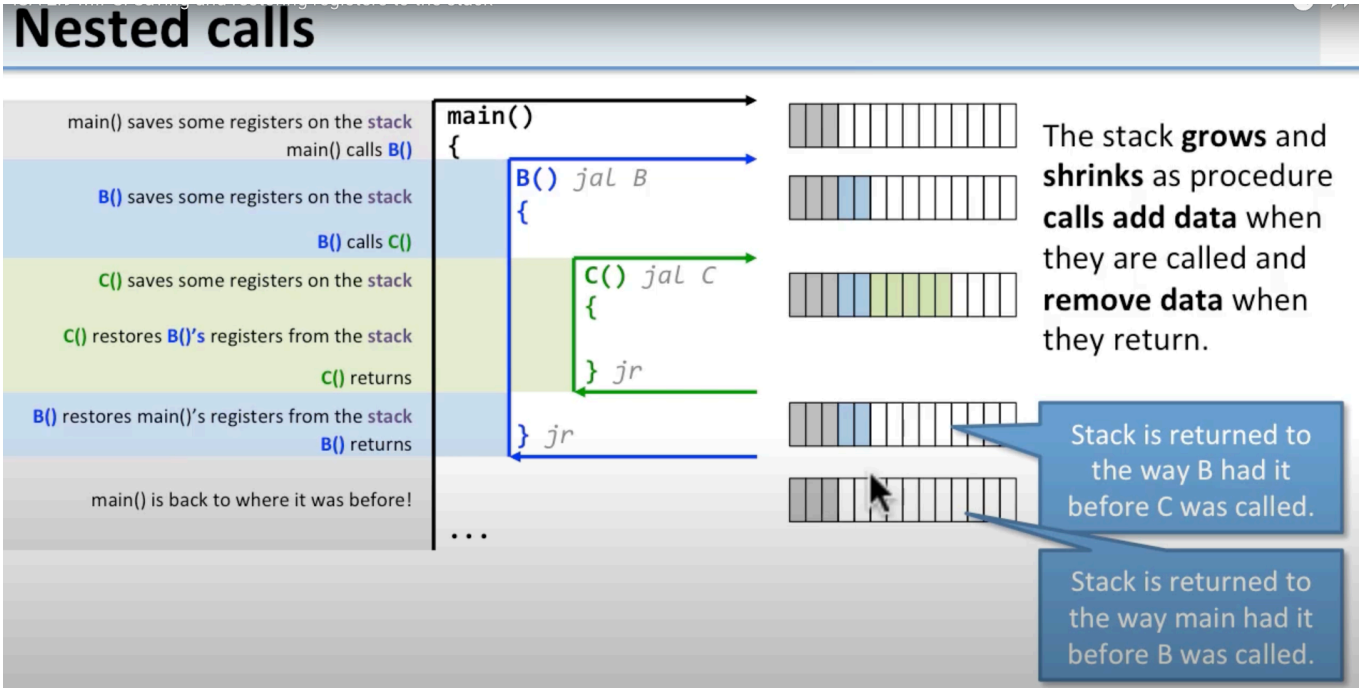
We dont know how much memory is needed to allocate in both stack or dynamic memory, so both will start at some point (**stack** - starts at high memory address and go down / **dynamic memory** - starts at low memory address and goes up)




sum of byters allocated for both stack and dynmaic memory must not be more than the difference between the starting address for both ()

**Stack pointer** - points to where the stack is at (end of stack) as it goes downwards

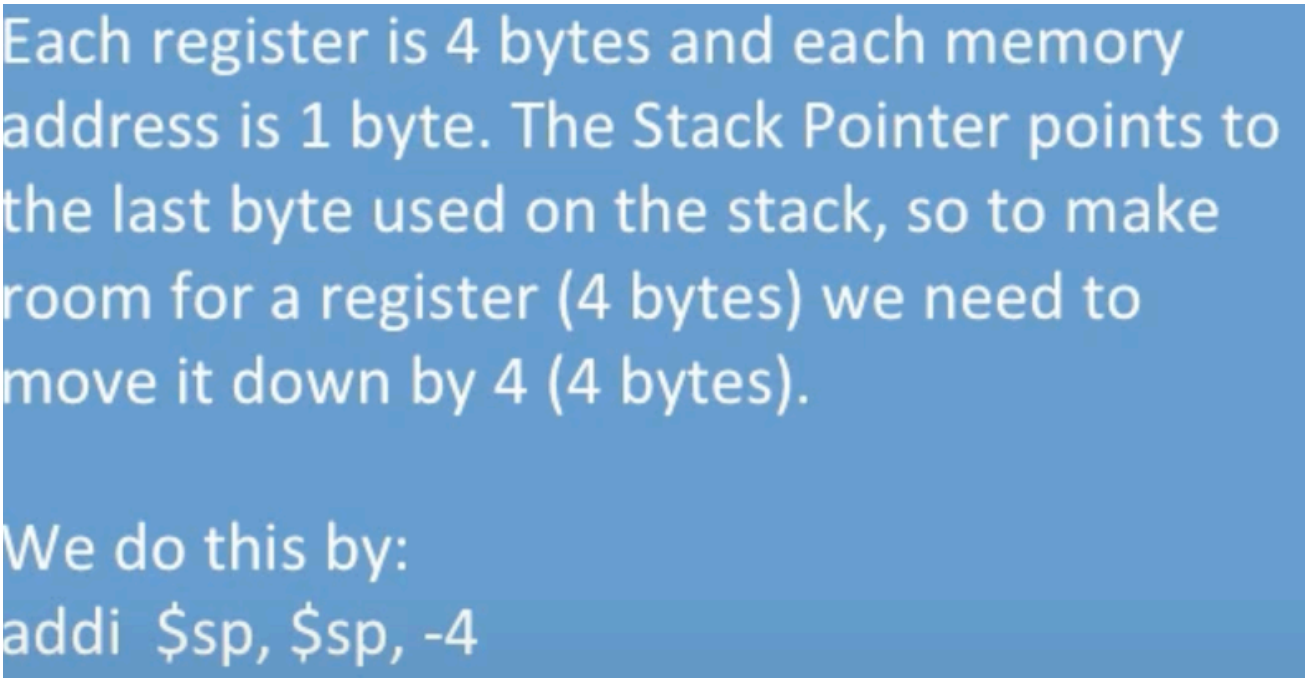
**Nested Calls Diagram**

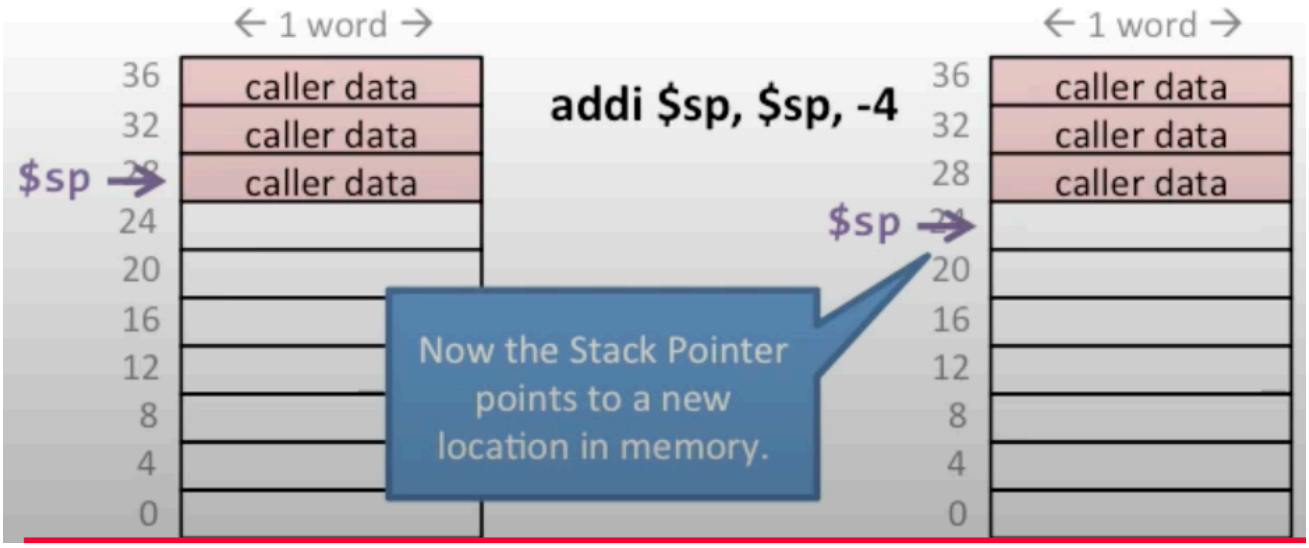



## How do you save/restore from stack

### save

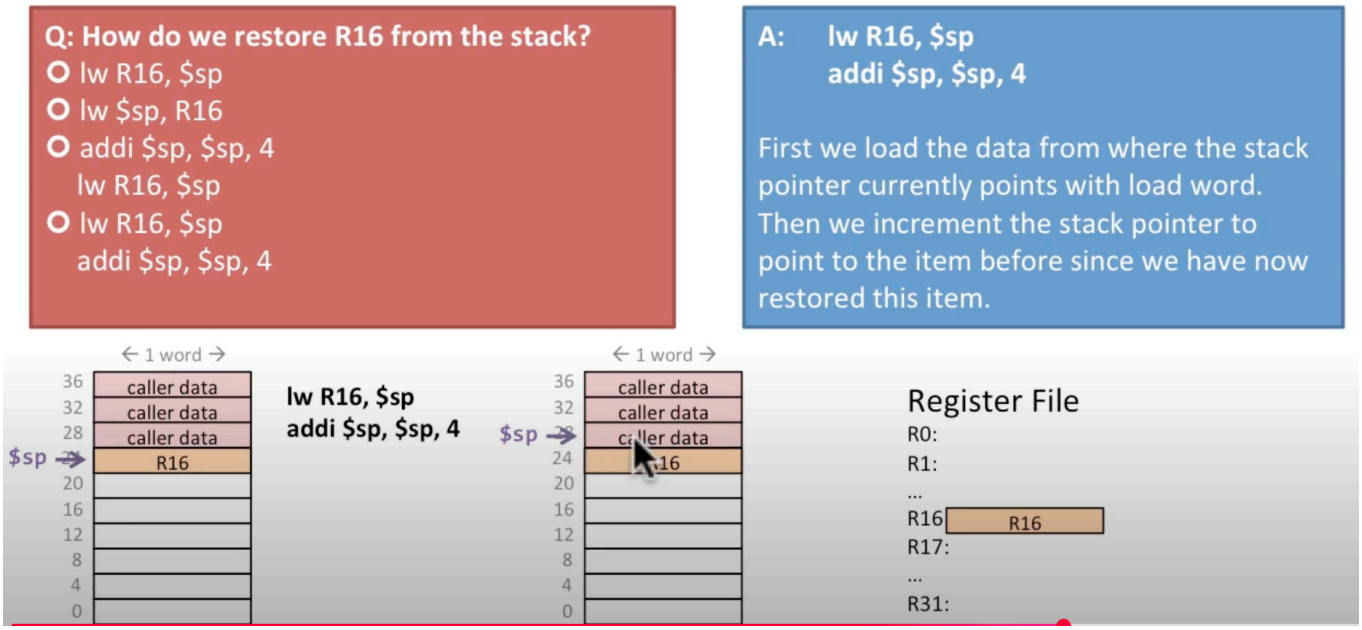
- how do you store a register in stack? general case:






(stack gets decremenetd to point to a new location in memory)

**restore**




## What do we need to store in the stack frame?

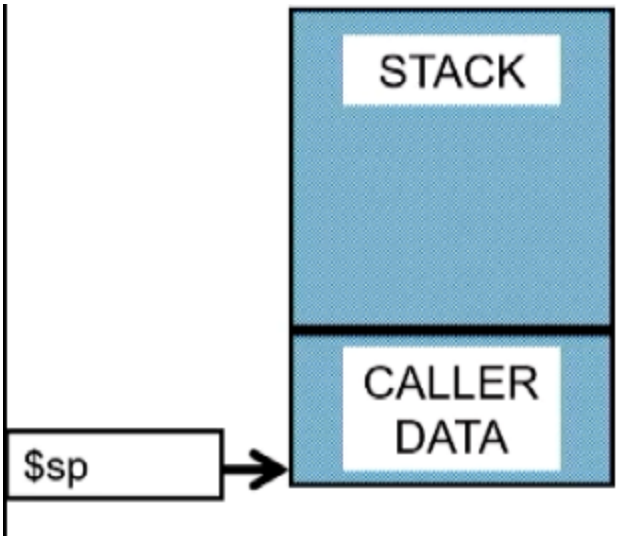
- **return address** `$ra`
  - When a function makes nested calls, it needs to save its own return address before the jal instruction overwrites `$ra` with the new return address for the nested call. Hence we need to store the * *`$ra`* in the stack
- **frame pointer `$fp**
  - We need to restore the value of frame pointer after a function call terminates
  - It provides a stable reference point to access stack items even as the stack pointer changes during function execution. Each function needs to restore the previous frame pointer when it returns.

    - Frame pointer always points to beginning of current function stack region (where the stack region starts for the current function call)`
- **preivous arguments ( `$a0` - `$a3` )**
    - Previous arguments might be used after functions have been called and that different arguments can be used when performing function calls inside functions as different functions expect different number of arguments ( `$a0` - `$a3` )
- **all saved registers used by caller/callee**
    - Caller-saved registers ( `$t0` - `$t9` ) must be saved by the caller if their values are needed after a function call
    - Callee-saved registers ( `$s0` - `$s7` ) must be saved and restored by the callee if it uses them
- **local variables**
    - local variables tend to be stored in stack (they can be stored in registers but there is a case there is not enough reigsters to store all local variables - thus store on stack)

## How does the stack work for function calls?

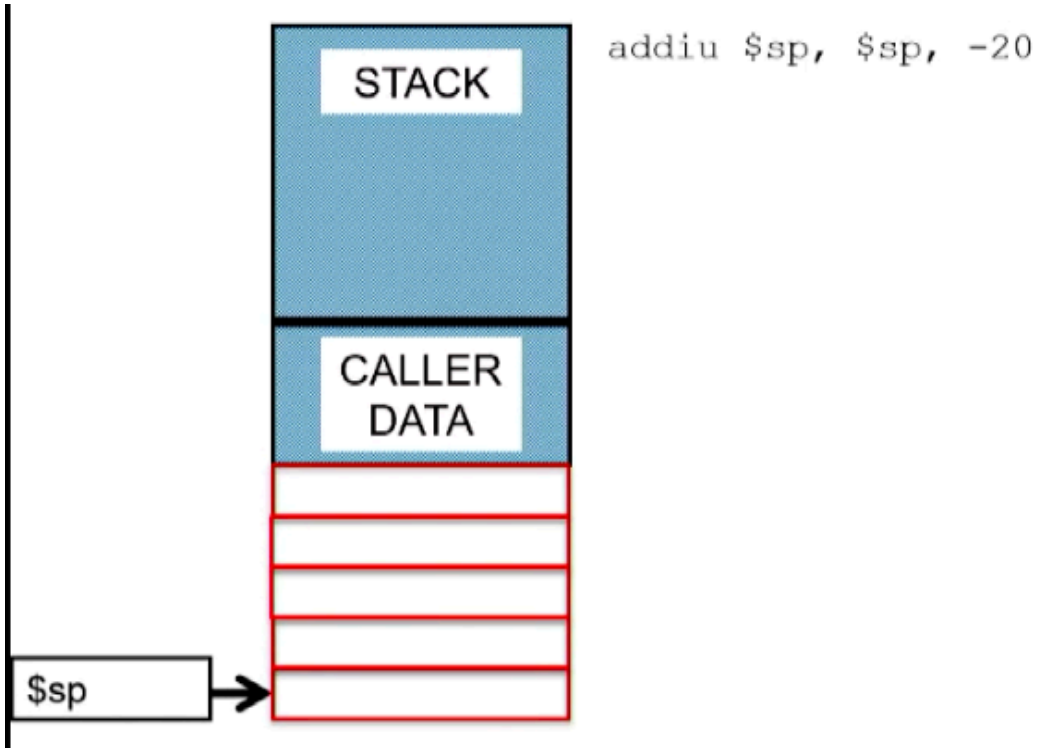
given we have a one argument function `int f(int y)`

**stack at beginning of function:**




stack pointer points at the data

**stack when calling a function**

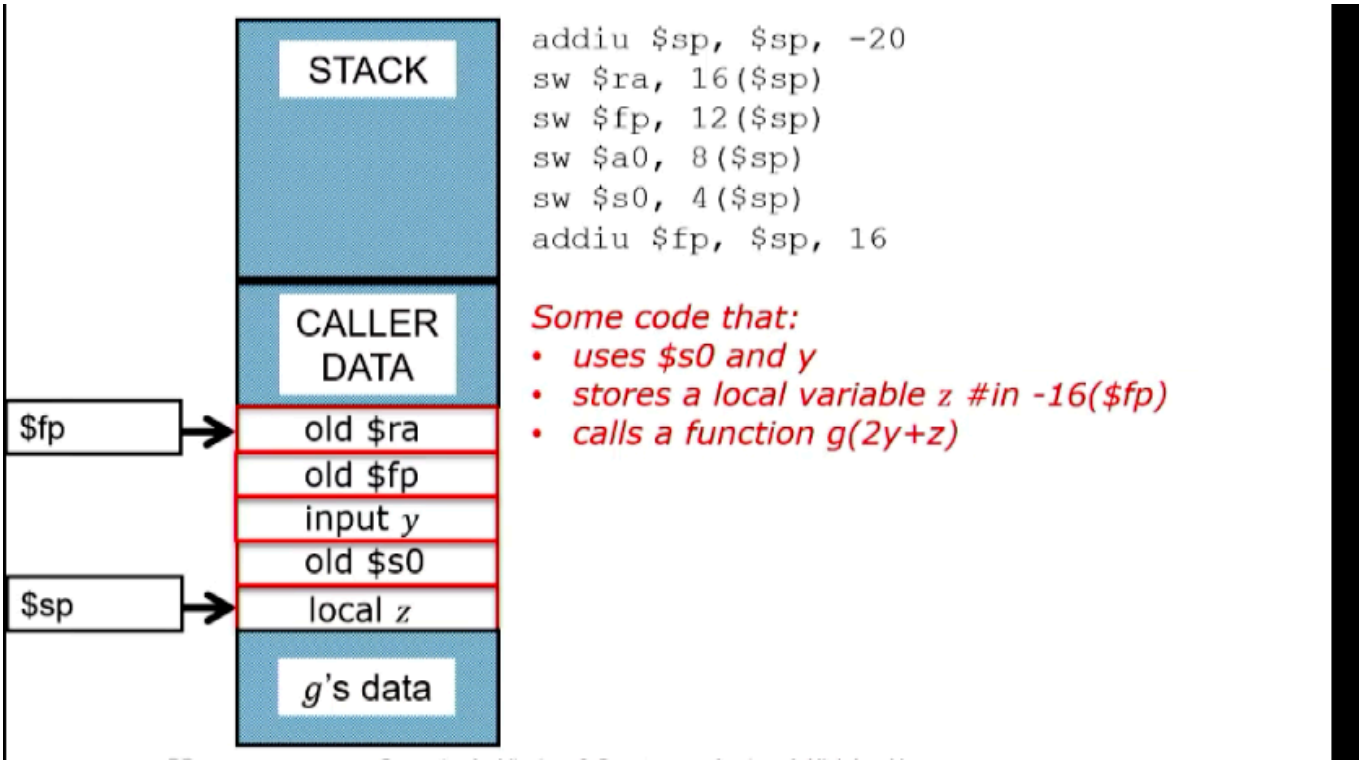



**a new stack frame is created**

in the example, stack pointer gets decremented by 20

- allocate 20 bytes on teh stack, it contains memory for variables/registers for that function call
- **stack pointer gets decremented** further when there are more nested function calls made, to allocate memory for that function call
- when a function starts, it shouldn't access the memory region for the previous functions calls. it should allocate its own memory for its variables, registers, etc.

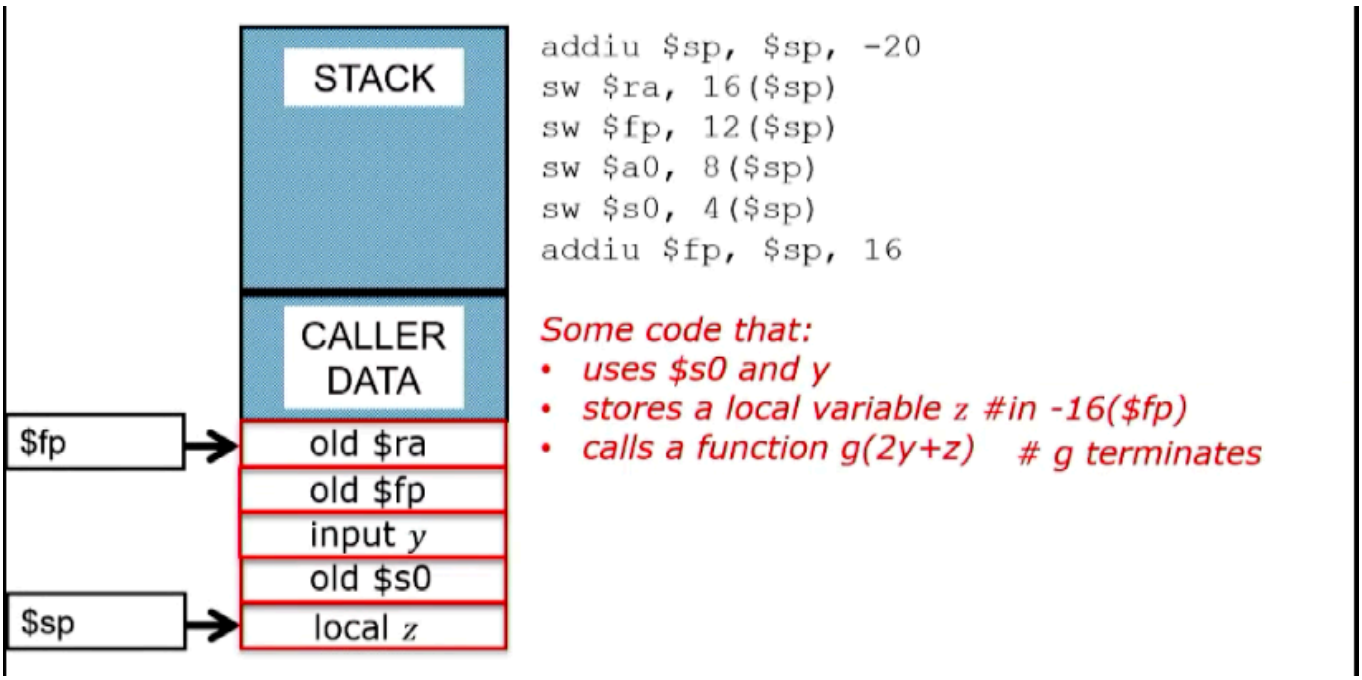
**Example when calling a function inside function f**




The process repeats when calling the function `g` where a stack frame is created for function `g` (i.e. increment stack pointer/frame pointer/ store g's data consisting of return address of g, and arguments of g local variables of g). `g` will allocate some memory on stack where it shouldnt touch the memory space of the previous function call.

stack pointer will be decremeted, followed by storing $ra, $fp, arguments, saved register values, and local variables, and then frame pointer will be incremented to point to start of the memory space allocated for function `g`

**After function g terminates:**



stack pointer and frame pointer gets decremented (according to convention, function `g` is responible for reverrting the values of both stack and frame pointer but doesn't modify these values) - hence the need to store the values in the stack frame as we would need to use the original values before calling a nested function, after that function terminates

When a function executes:

1. It decrements the stack pointer (`$sp`) to allocate space for its stack frame
2. It saves the current frame pointer (`$fp`) on the stack
3. It sets its own frame pointer to the new frame boundary (typically `$fp = $sp + offset`)
4. When calling a nested function, the current function's context must be preserved

Then when returning:

1. The function must restore `$sp` and `$fp` to their original values
2. This is why we save the original `$fp` value on the stack - so we can restore it
3. By convention, each function is responsible for cleaning up its own stack frame

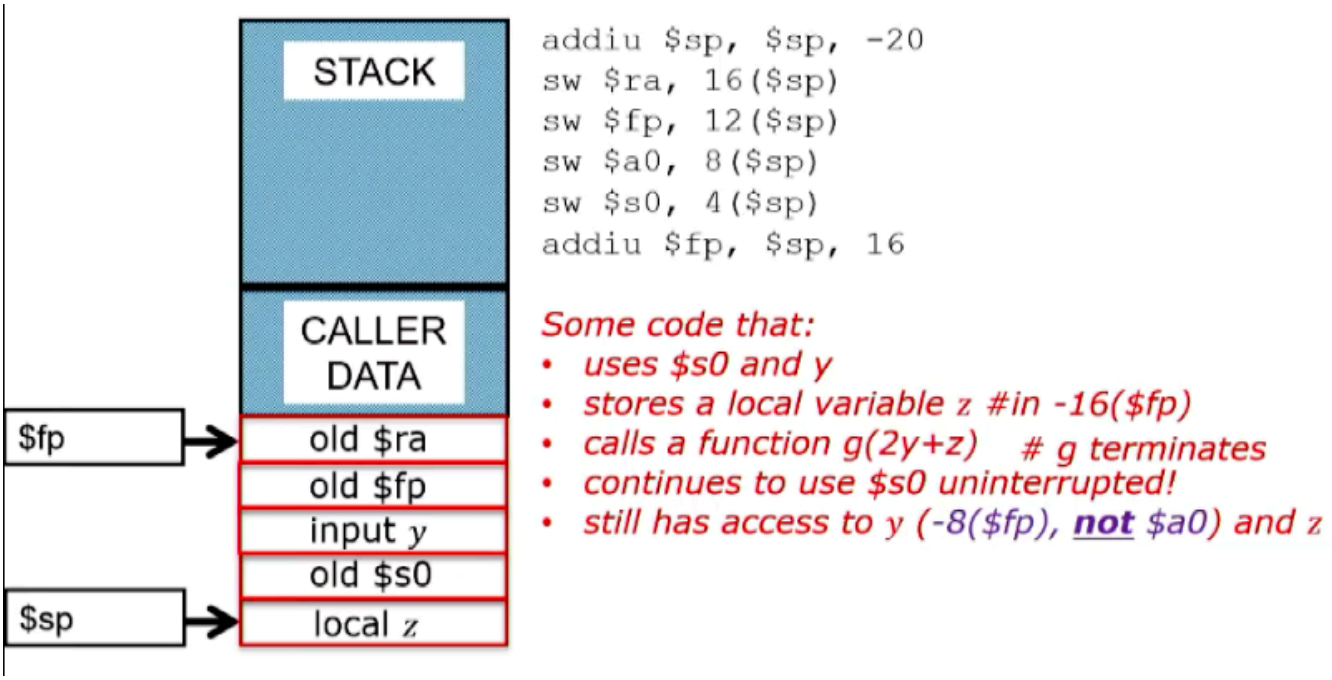
The critical part is that each function follows a consistent protocol:

- Save state when entering
- Restore state when exiting
- Never modify parent function stack frames

This approach creates a "chain" of stack frames, where each function can find its caller's context by following the saved frame pointers. This is particularly important because:

1. The function `g` doesn't know (and shouldn't need to know) how much stack space `f` allocated
2. When `g` returns, `f` needs to continue execution with its original context intact
3. Without storing the original `$fp` and `$sp` values, `g` would have no way to properly return to `f`'s context

So yes, saving these pointers is essential for maintaining the proper function call hierarchy.

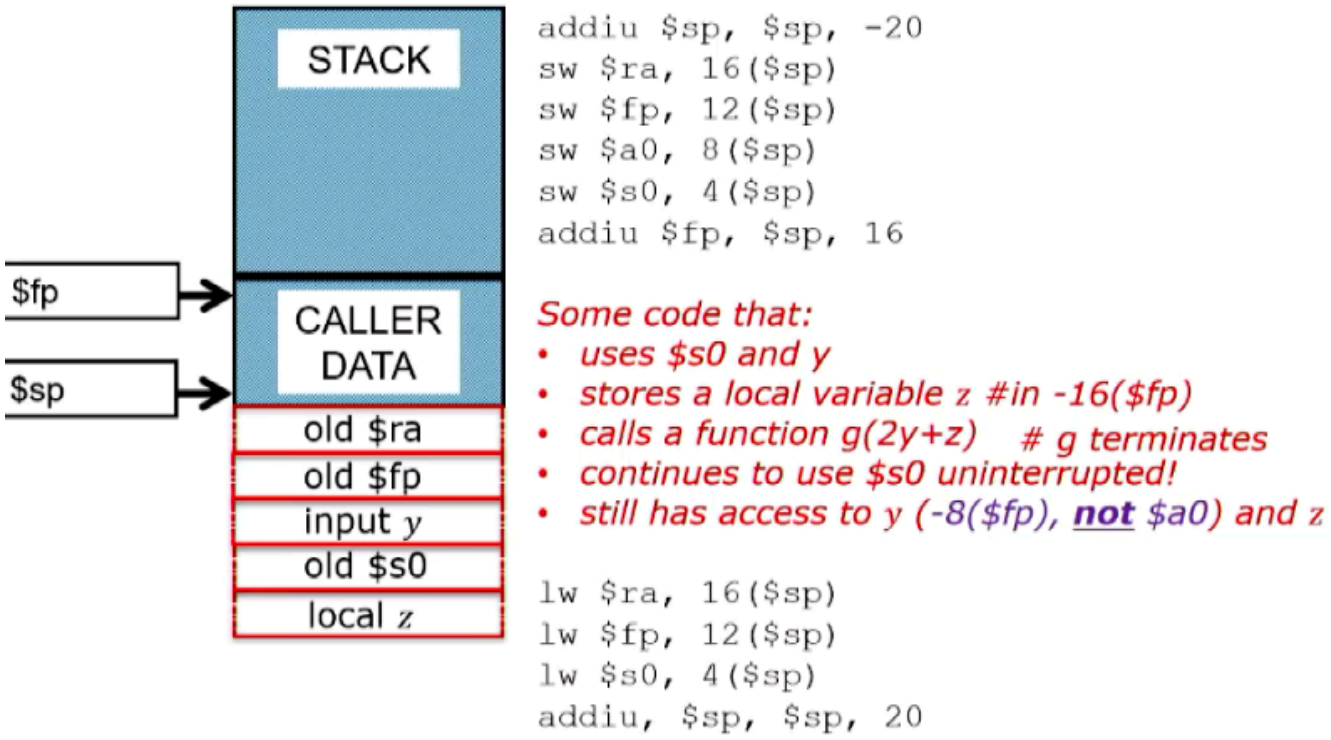


value in $s0 is reverted to what is stored at top of stack (for function `f`)

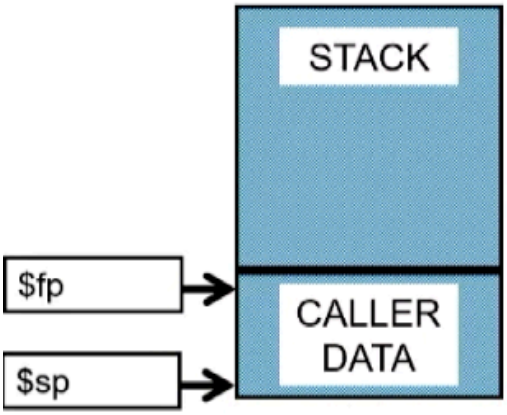
based on diagram, function f still has access `y` but it is stored in the stack frame instead of `$a0`, as `$a0` was used to encode the value passed to `g` (hence why we have to store arguments onto stack, so they can be accessed, i.e. 8 bytes away from frame pointer)

after g terminates, we need to restore all the register values from memory like the previous frame pointer, return address register and the saved registers (as the values in saved registers are preserved) - we need to know the offset after allocating memory for stack frame that was created after calling a function

**After function `f` terminates**




stack pointer need to increment by 20 bytes (to go back previous location) to point to end of stack at caller data (the stack frame for caller) after f terminates to free memory space allocated.




```
addiu $sp, $sp, -20
sw $ra, 16($sp)
sw $fp, 12($sp)
sw $a0, 8($sp)
sw $s0, 4($sp)
addiu $fp, $sp, 16
```

*Some code that:*

- *uses $s0 and y*
- *stores a local variable z #in -16($fp)*
- *calls a function g(2y+z)  # g terminates*
- *continues to use $s0 uninterrupted!*
- *still has access to y (-8($fp), **not** $a0) and z*

```
lw $ra, 16($sp)
lw $fp, 12($sp)
lw $s0, 4($sp)
addiu, $sp, $sp, 20
```

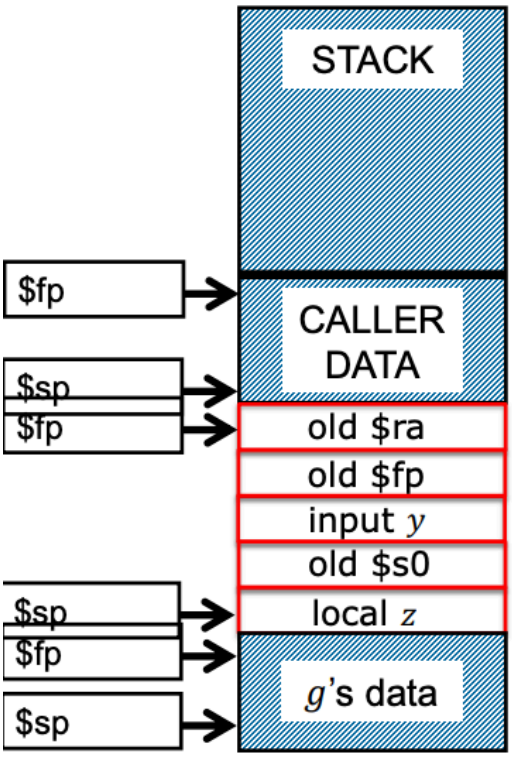
If function that called `f` calls another function, that other callee function will reuse that same memory space allocated prior for `f`

Only active functions calls are stored in the call stack (terminated functions have their memory freed in stack)

**Summary**

During compilation, compiler will determine which registers to store in the stack when

performing nested function calls




```
addiu $sp, $sp, -20
sw $ra, 16($sp)
sw $fp, 12($sp)
sw $a0, 8($sp)
sw $s0, 4($sp)
addiu $fp, $sp, 16
```

*Some code that:*

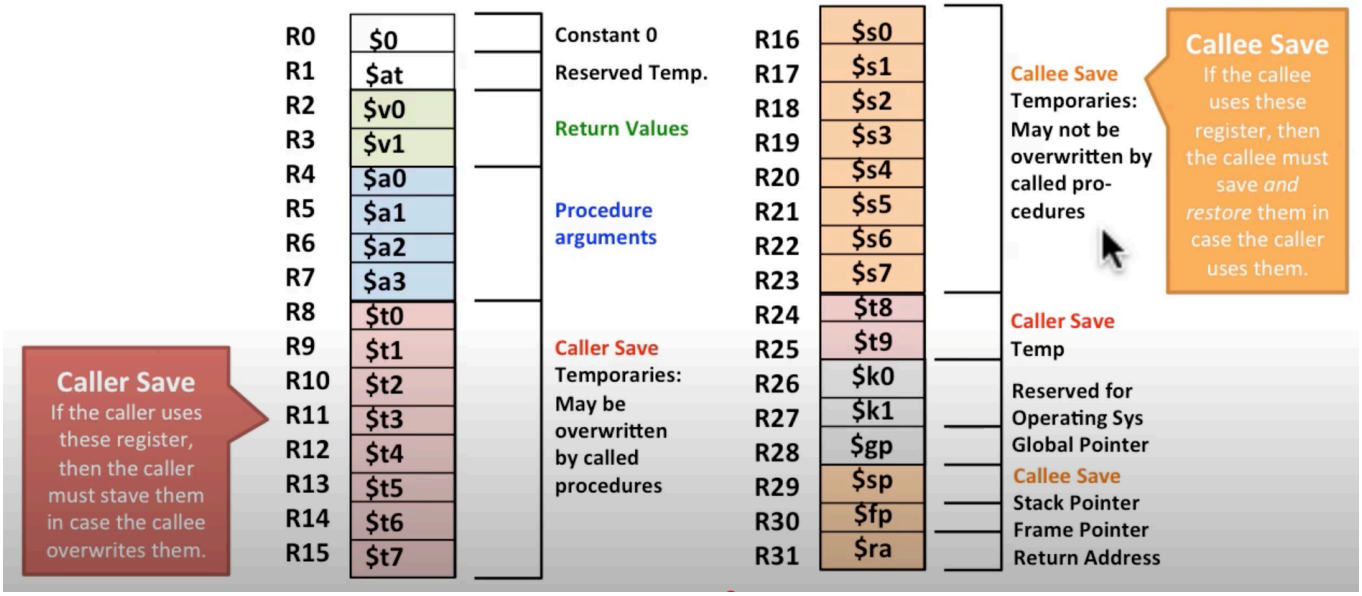
- *uses $s0 and y*
- *stores a local variable z #in -16($fp)*
- *calls a function g(2y+z)  # g terminates*
- *continues to use $s0 uninterrupted!*
- *still has access to y (-8($fp), **not** $a0) and z*

```
lw $ra, 16($sp)
lw $fp, 12($sp)
lw $s0, 4($sp)
addiu, $sp, $sp, 20
```

Benefit of frame pointer:

- a function might not know all of the things it needs to write to stack at beginning of the function
- frame pointer remains if function allocates mroe space (as stack pointer decrements to reflect the allocation of memory)
- we dont have to keep track of how many bytes were allocated to know where inputs (arguments) can be extracted - given we have a frame pointer and we know the offset
- if you dont have frame pointer, you need to keep track of how many bytes allocated to find out the address of argument (if use offset + frame pointer, the address is known during assembly time)

## Who saves what?



**Allocating new space for new data on stack - textbook diagram**

## Allocating Space for New Data on the Stack

The final complexity is that the stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures. The segment of the stack containing a procedure's saved registers and local variables is called a **procedure frame** or **activation record**. Figure 2.12 shows the state of the stack before, during, and after the procedure call.

**procedure frame** Also called **activation record**. The segment of the stack containing a procedure's saved registers and local variables.

Some MIPS software uses a **frame pointer** (`$fp`) to point to the first word of the frame of a procedure. A stack pointer might change during the procedure, and so references to a local variable in memory might have different offsets depending on where they are in the procedure, making the procedure harder to understand. Alternatively, a frame pointer offers a stable base register within a procedure for local memory-references. Note that an activation record appears on the stack whether or not an explicit frame pointer is used. We've been avoiding using `$fp` by avoiding changes to `$sp` within a procedure: in our examples, the stack is adjusted only on entry and exit of the procedure.

**frame pointer** A value denoting the location of the saved registers and local variables for a given procedure.

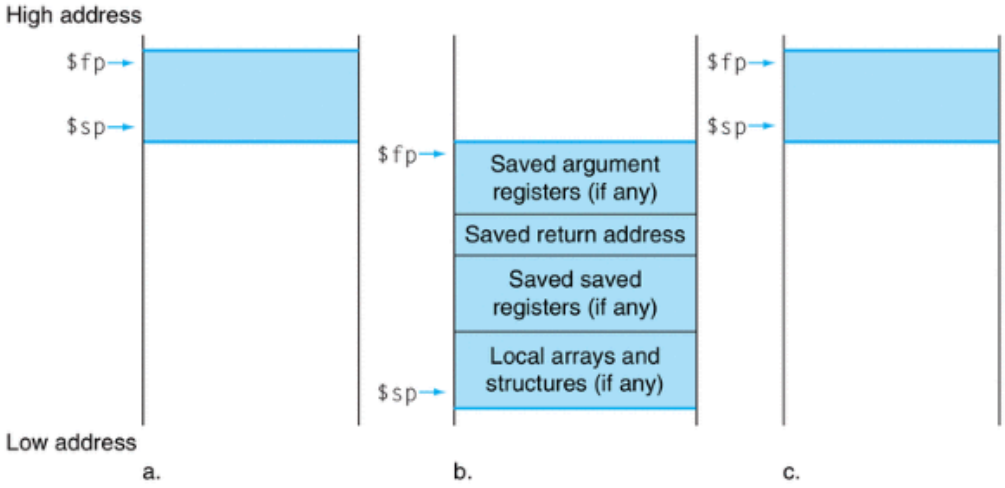



**FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call.** The frame pointer (`$fp`) points to the first word of the frame, often a saved argument register, and the stack pointer (`$sp`) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it's easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by *not* setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in `$sp` on a call, and `$sp` is restored using `$fp`. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

**Example - abs_diff function with stack memory allocation**

```
# Calculating absolute difference without infinite loop !
my_program:
        addi $sp, $sp, -8 # Move the stack pointer.
        sw $ra, 4($sp)    # Store return address.
        sw $fp, 0($sp)    # Store frame pointer.
        addi $fp, $sp, 4  # Set new frame pointer.

        la $t1, vals      # Load address of data into $t1.
        lw $a0, 0($t1)    # Load A from memory into $a0.
        lw $a1, 4($t1)    # Load B from memory into $a1.
        jal abs_diff      # Jump and link to 'abs_diff' function.
        la $t1, vals      # Load address of data into $t1 again.

        sw $v0, 8($t1)    # Write result and return.

        lw $ra, 4($sp)    # Restore return address.
        lw $fp, 0($sp)    # Restore frame pointer.
        addi $sp, $sp, 8  # Restore stack pointer position.

        jr $ra            # Return from program.
```

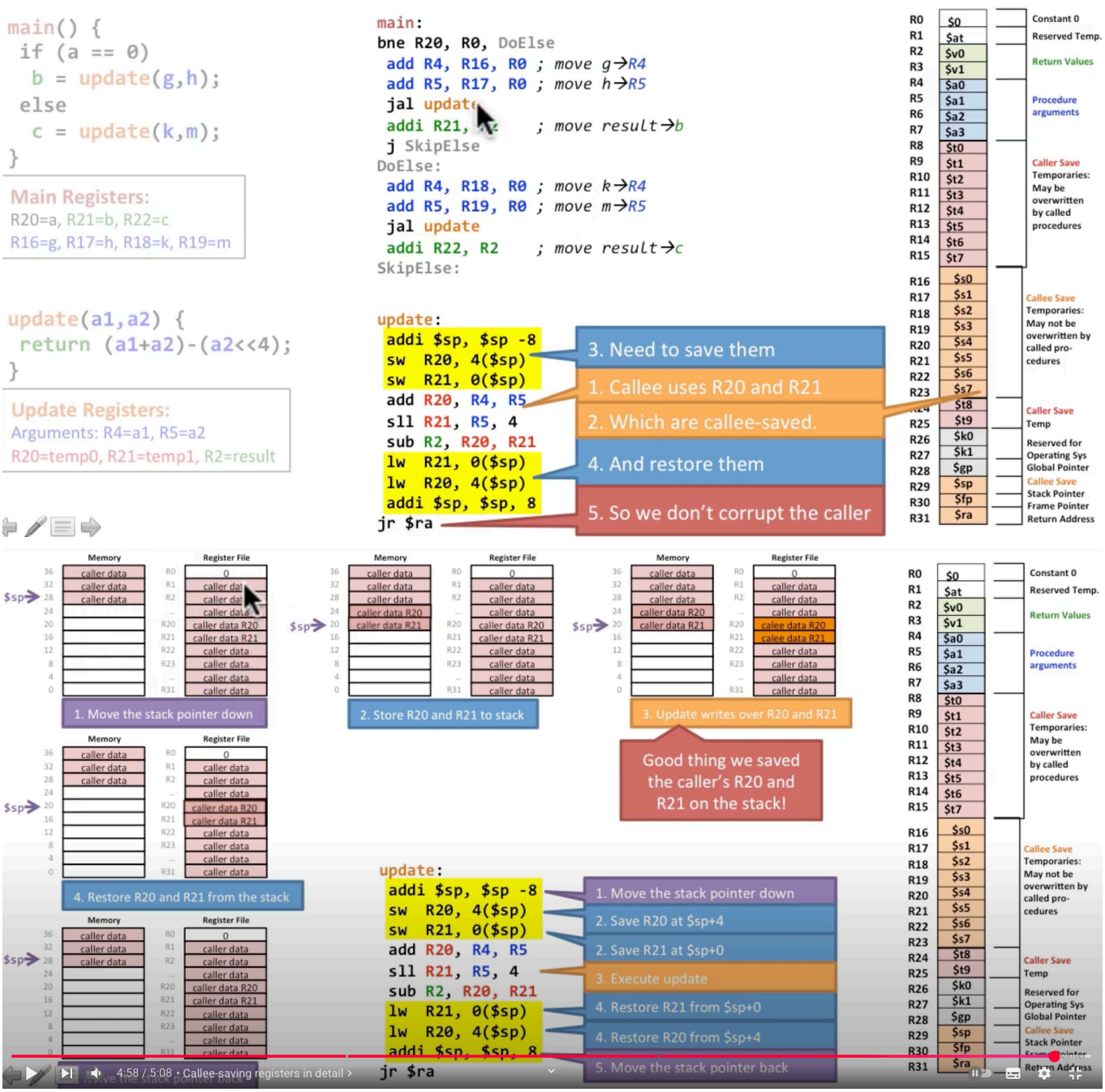
after calling the function `abs_diff`, need to load address of vals again
after storing the result in 3rd element of vals, have to load the return address, frame pointer from memory and restore the stack pointer position to free memory

as the registers used are temporary, caller has to save/restore the registers (if saved registers, the callee does that)

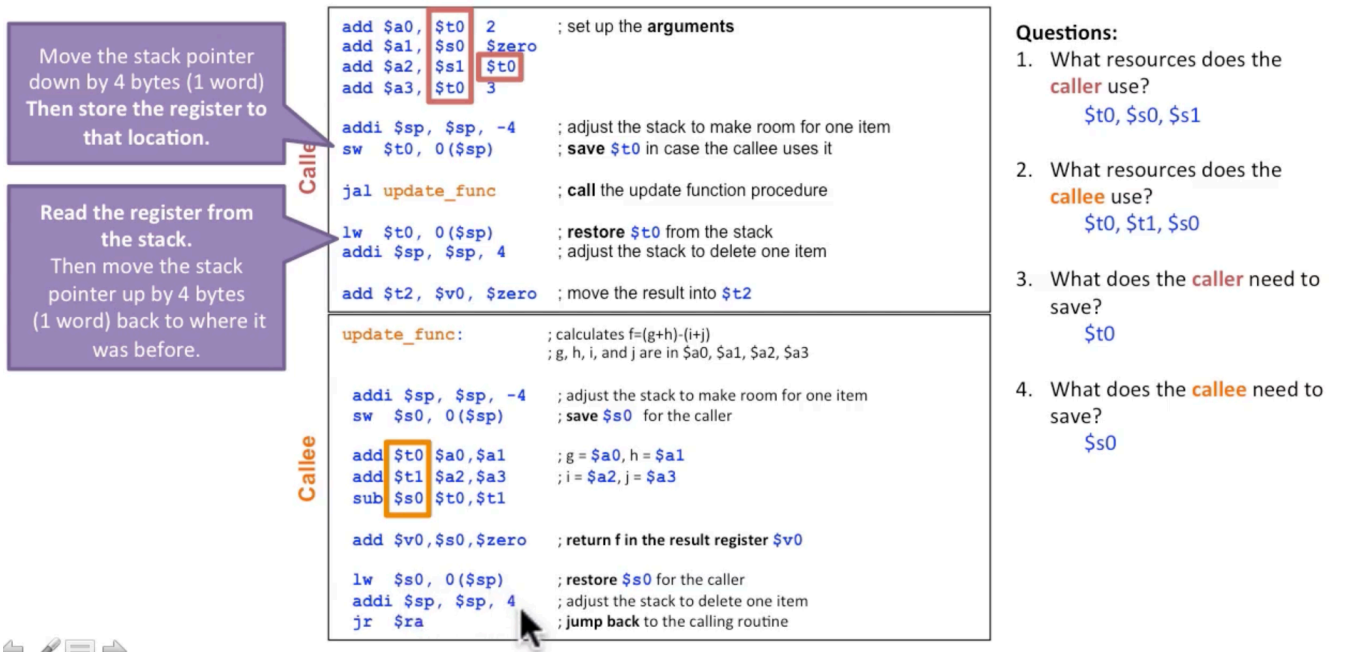
## YouTube example

Saving/Restore registers by Callee

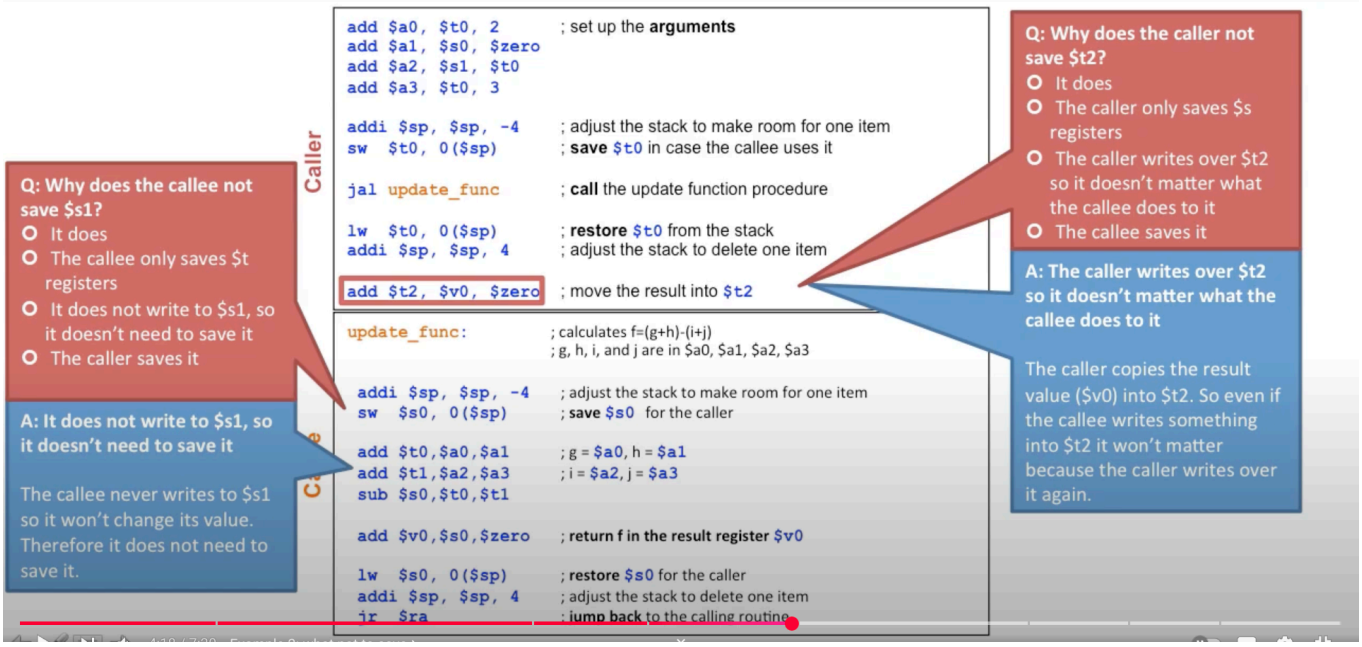
# Example 1: saving registers



# Example 2: saving to the stack




You don't need to save/restore registers that are not used (callee needs to save/restore saved registers it uses, caller needs to save/restore temporary registers that it uses)




## Stack example from Textbook

The next step is to save the registers used by the procedure. The C assignment statement in the procedure body is identical to the example on page 79, which uses two temporary registers. Thus, we need to save three registers: `$s0`, `$t0`, and `$t1`. We "push" the old values onto the stack by creating space for three words (12 bytes) on the stack and then store them:

```
addi $sp, $sp, -12    # adjust stack to make room for 3 items
sw  $t1, 8($sp)       # save register $t1 for use afterwards
sw  $t0, 4($sp)       # save register $t0 for use afterwards
sw  $s0, 0($sp)       # save register $s0 for use afterwards
```

Figure 2.10 shows the stack before, during, and after the procedure call.

The next three statements correspond to the body of the procedure, which follows the example on page 79:

```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)
```

To return the value of `f`, we copy it into a return value register:

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

Before returning, we restore the three old values of the registers we saved by "popping" them from the stack:

```
lw  $s0, 0($sp)  # restore register $s0 for caller
lw  $t0, 4($sp)  # restore register $t0 for caller
lw  $t1, 8($sp)  # restore register $t1 for caller
addi $sp,$sp,12  # adjust stack to delete 3 items
```

The procedure ends with a jump register using the return address:

```
jr  $ra    # jump back to calling routine
```

In the previous example, we used temporary registers and assumed their old values must be saved and restored. To avoid saving and restoring a register whose value is never used, which might happen with a temporary register, MIPS software separates 18 of the registers into two groups:

- `$t0`–`$t9`: ten temporary registers that are *not* preserved by the callee (called procedure) on a procedure call
- `$s0`–`$s7`: eight saved registers that must be preserved on a procedure call (if used, the callee saves and restores them)

This simple convention reduces register spilling. In the example above, since the caller does not expect registers `$t0` and `$t1` to be preserved across a procedure call,

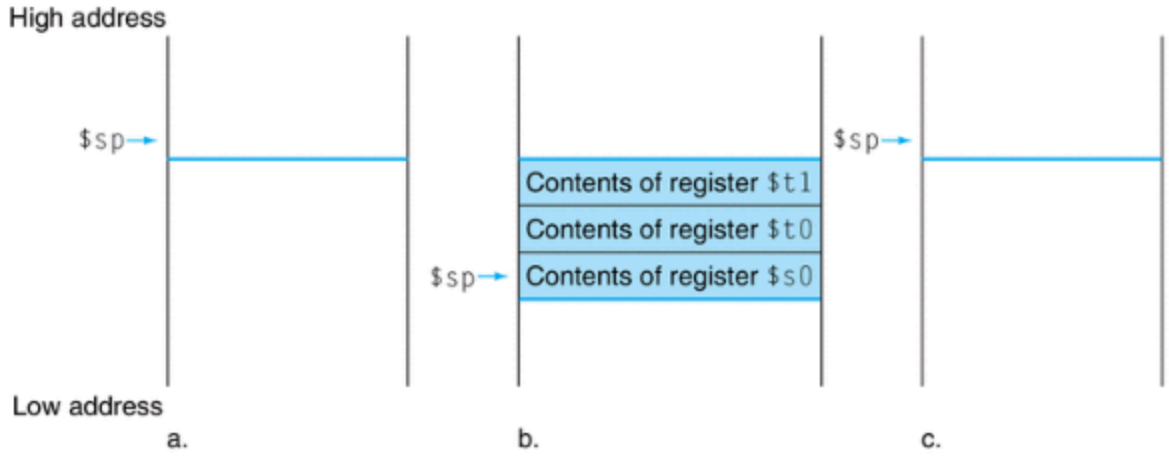



**FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call.** The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Nested procedures

Procedures that do not call others are called *leaf* procedures. Life would be simple if all procedures were leaf procedures, but they aren't. Just as a spy might employ other spies as part of a mission, who in turn might use even more spies, so do procedures invoke other procedures. Moreover, recursive procedures even invoke "clones" of themselves. Just as we need to be careful when using registers in procedures, more care must also be taken when invoking nonleaf procedures.

For example, suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register `$a0` and then using `jal A`. Then suppose that procedure A calls procedure B via `jal B` with an argument of 7, also placed in `$a0`. Since A hasn't finished its task yet, there is a conflict over the use of register `$a0`. Similarly, there is a conflict over the return address in register `$ra`, since it now has the return address for B. Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.

One solution is to push all the other registers that must be preserved onto the stack, just as we did with the saved registers. The caller pushes any argument registers (`$a0–$a3`) or temporary registers (`$t0–$t9`) that are needed after the call. The callee pushes the return address register `$ra` and any saved registers (`$s0–$s7`) used by the callee. The stack pointer `$sp` is adjusted to account for the number of registers placed on the stack. Upon the return, the registers are restored from memory and the stack pointer is readjusted.

## Recursive Procedure in MIPS

### Compiling a Recursive C Procedure, Showing Nested Procedure Linking

**EXAMPLE**

Let's tackle a recursive procedure that calculates factorial:

```
int fact (int n)
{
    if (n < 1) return (1);
          else return (n * fact(n - 1));
}
```

What is the MIPS assembly code?

**ANSWER**

The parameter variable `n` corresponds to the argument register `$a0`. The compiled program starts with the label of the procedure and then saves two registers on the stack, the return address and `$a0`:

```
fact:
    addi   $sp, $sp, -8  # adjust stack for 2 items
    sw     $ra, 4($sp)   # save the return address
    sw     $a0, 0($sp)   # save the argument n
```

The first time `fact` is called, `sw` saves an address in the program that called `fact`. The next two instructions test whether `n` is less than 1, going to `L1` if $n \geq 1$.

```
      slti   $t0,$a0,1      # test for n < 1
      beq    $t0,$zero,L1   # if n >= 1, go to L1
```

If `n` is less than 1, `fact` returns 1 by putting 1 into a value register: it adds 1 to 0 and places that sum in `$v0`. It then pops the two saved values off the stack and jumps to the return address:

```
    addi   $v0,$zero,1 # return 1
    addi   $sp,$sp,8   # pop 2 items off stack
    jr     $ra         # return to caller
```

Before popping two items off the stack, we could have loaded `$a0` and `$ra`. Since `$a0` and `$ra` don't change when `n` is less than 1, we skip those instructions.

If `n` is not less than 1, the argument `n` is decremented and then `fact` is called again with the decremented value:

```
L1: addi $a0,$a0,-1   # n >= 1: argument gets (n - 1)
    jal  fact         # call fact with (n - 1)
```

> The next instruction is where `fact` returns. Now the old return address and old argument are restored, along with the stack pointer:
>
> ```
> lw   $a0, 0($sp)   # return from jal: restore argument n
> lw   $ra, 4($sp)   # restore the return address
> addi $sp, $sp, 8   # adjust stack pointer to pop 2 items
> ```
>
> Next, the value register `$v0` gets the product of old argument `$a0` and the current value of the value register. We assume a multiply instruction is available, even though it is not covered until Chapter 3:
>
> ```
> mul  $v0,$a0,$v0   # return n * fact (n - 1)
> ```
>
> Finally, `fact` jumps again to the return address:
>
> ```
> jr   $ra           # return to the caller
> ```

## Example exam question

Q1a. As a reminder, when implementing pseudo-instructions, the assembler may modify $1. Assume that a MARS instruction X is followed by instruction Y. Which of the following is correct?

A) We may not assume that $1 is unchanged, regardless of whether X is a pseudo-instruction or a native instruction.
B) We may assume that $1 is unchanged if X is a `blt` instruction.
C) We may assume that $1 is unchanged if X is a `slt` instruction.
D) We may assume that $1 is unchanged if X is a `syscall` instruction.

assembler is allowed to modify value of `$1` ( `$at` ) without restoring it when ONLY) executing pseudo instructions

a. false - $1 only gets changed when executing pseudo instructions
b. false - `blt` is a pesudo instruction so $1 can be modifeid (cant assume it will be unchanged)
**c. true - `slt` is a native instruction so can assume that $1 won't be unchanged**
d. false

tip: you have to identify which instruction is native or pseudo e.g. `blt` is a pseudo instruction, according to the reference card, or can use MARS and see if that instruction gets translated to native (the reference card does not have all the native instructions)

Q1b. As a reminder, registers $k0 and $k1 are used by the kernel. Assume that a MARS instruction X is followed by instruction Y.

A) We may not assume that $k1 is unchanged, regardless of whether X is a pseudo-instruction or a native instruction.
B) We may assume that $k1 is unchanged if X is a `blt` instruction.
C) We may assume that $k1 is unchanged if X is a `slt` instruction.
D) We may assume that $k1 is unchanged if X is a `syscall` instruction.

**a. true - kernel can be invoked when there is an interrupt at any time which can alter values in $k0 and $k1**
b. false
c. false
d. false

Q1c. As a reminder, it is the *callee's* responsibility to maintain the values of the saved registers. Which of the following is correct about writing a function `f`?

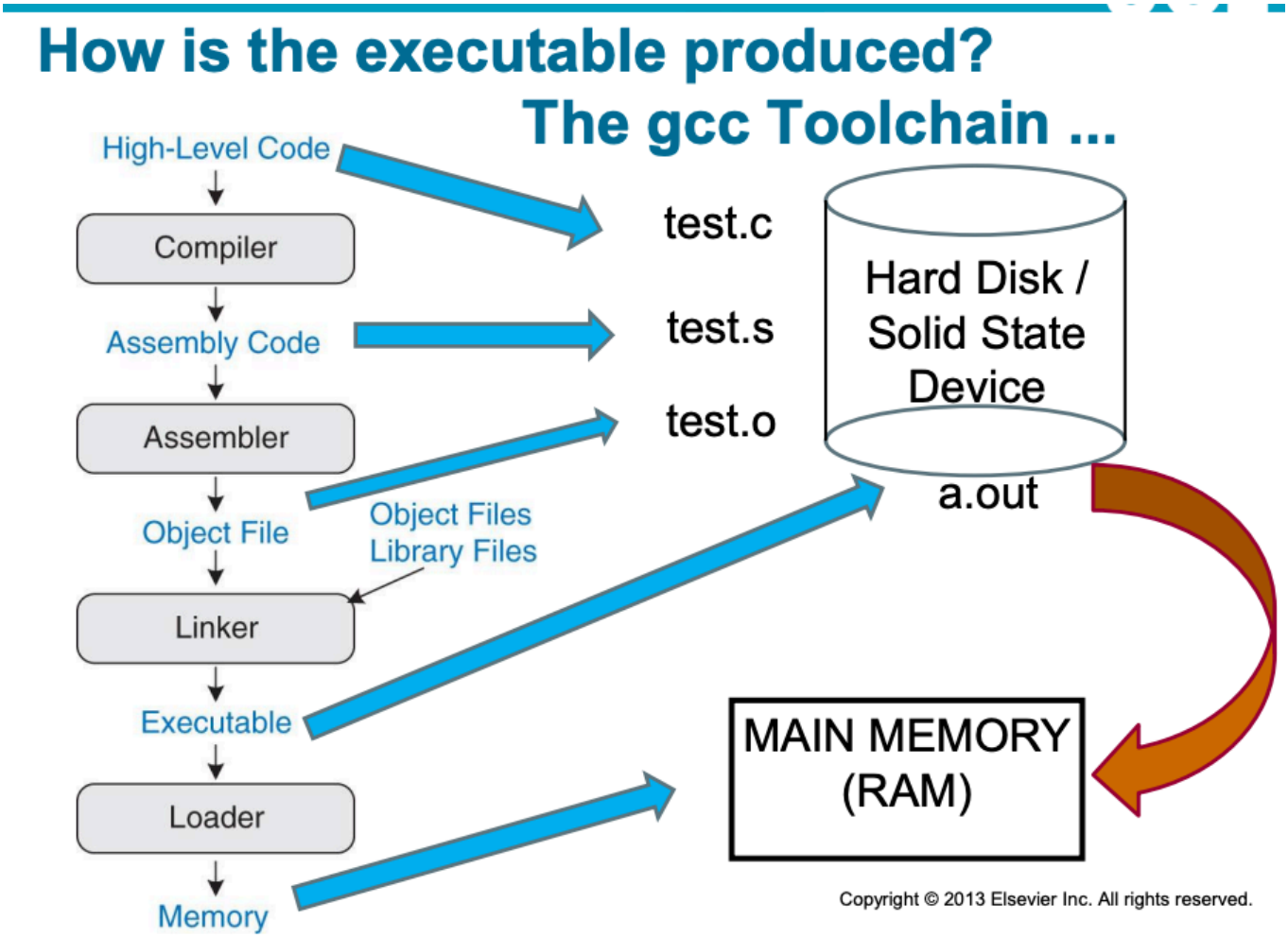
A) We may not modify the value of $s0 without restoring it, regardless of whether we call another function.
B) We may modify the value of $s0 if our function does not call another function.
C) We may only modify the value of $s0 if the only function call is recursively to `f`.
D) We may only modify the value of $s0 without restoring it if when `f` returns its value would be in $t0.

are we allowed to modify the value of s0 without restoring it
**a. true. s0 is saved register as caller of F can assume that s0 won't be modfied** (we cannot modify the value in $s0 unless it is restored after nested function call is terminated - can be modified if we store $s0 value in the call stack)
b. false
c. false
d. false. whoever calls f has to assume that the value of $s0 will not expect to be in $t0

# Week 5 - System Calls / Basic Data Types

## Pre-Recorded (Translating / Starting a Program)

How is a C program translated and produced into an executable file?




**Compiler**:

- transforms C program into assembly language program (symbolic ofrm of what the machine understands)

**Assembler**:

- Assembles the assembly language program into an object module in machine language
- Can translate pseudo instructions (simplified instructions that are not actual native instructions) into native instructions that are understood by the machine
- Keeps track of labels used in branches and data transfer instructions in a symbol table

**Linker**:

- a systems program that combines independently assembled machine language programs (object files) and resolves all undefined labels into an executable file that can be run on a computer
- also known as link editor
- they can be linked statically or dynamically (dynamic link libraries), where **DLL** are library routines linked to a program during execution and part of the executable code

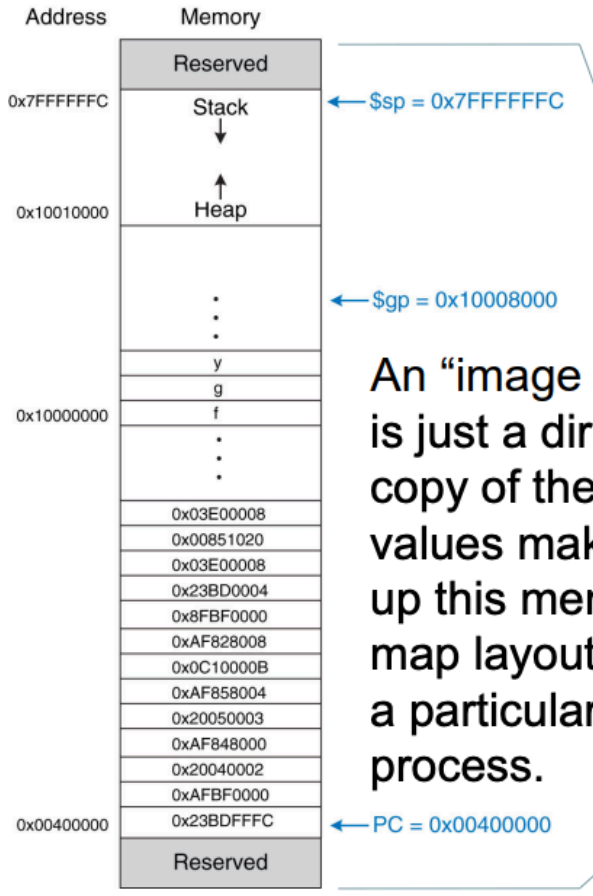
**Loader**:

- Places an object program (executable file on disk) in main memory so that it is ready to execute

# Binary Executable File on Disk

| Executable file header | Text Size | Data Size |
|---|---|---|
| | 0x34 (52 bytes) | 0xC (12 bytes) |
| **Text segment** | **Address** | **Instruction** |
| | 0x00400000 | 0x23BDFFFC |
| | 0x00400004 | 0xAFBF0000 |
| | 0x00400008 | 0x20040002 |
| | 0x0040000C | 0xAF848000 |
| | 0x00400010 | 0x20050003 |
| | 0x00400014 | 0xAF858004 |
| | 0x00400018 | 0x0C10000B |
| | 0x0040001C | 0xAF828008 |
| | 0x00400020 | 0x8FBF0000 |
| | 0x00400024 | 0x23BD0004 |
| | 0x00400028 | 0x03E00008 |
| | 0x0040002C | 0x00851020 |
| | 0x00400030 | 0x03E00008 |
| **Data segment** | **Address** | **Data** |
| | 0x10000000 | f |
| | 0x10000004 | g |
| | 0x10000008 | y |

```
addi $sp, $sp, –4
sw   $ra, 0($sp)
addi $a0, $0, 2
sw   $a0, 0x8000($gp)
addi $a1, $0, 3
sw   $a1, 0x8004($gp)
jal  0x0040002C
sw   $v0, 0x8008($gp)
lw   $ra, 0($sp)
addi $sp, $sp, –4
jr   $ra
add  $v0, $a0, $a1
jr   $ra
```

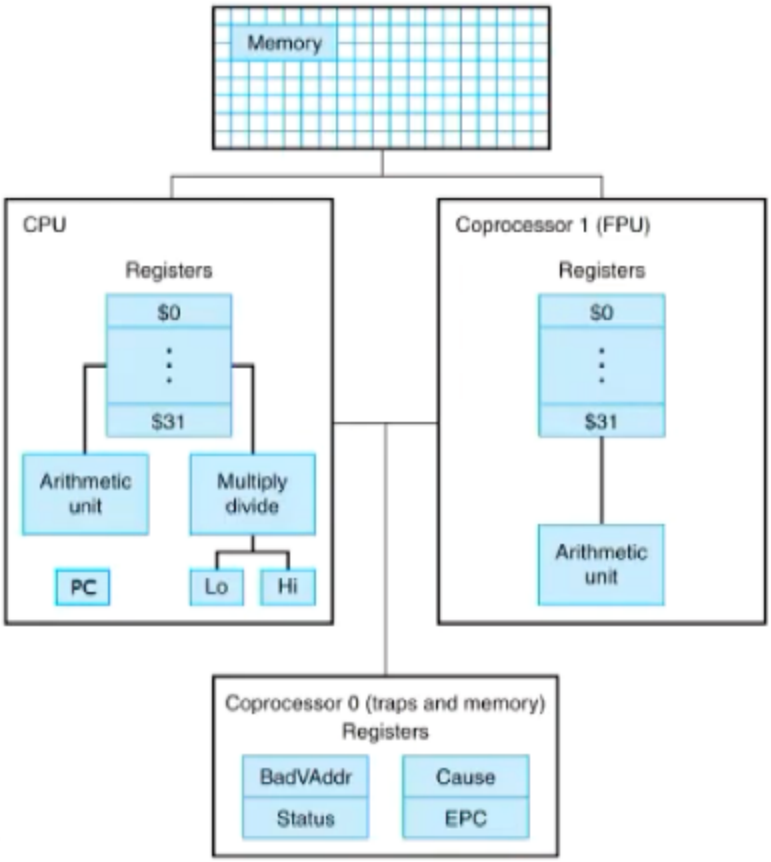

An "image file" is just a direct copy of the values making up this memory map layout for a particular process.

**The OS calls a "loader" program to load the executable file into RAM following the MIPS memory map ... then it is executed**

MAIN MEMORY (RAM)



## MIPS Coprocessors



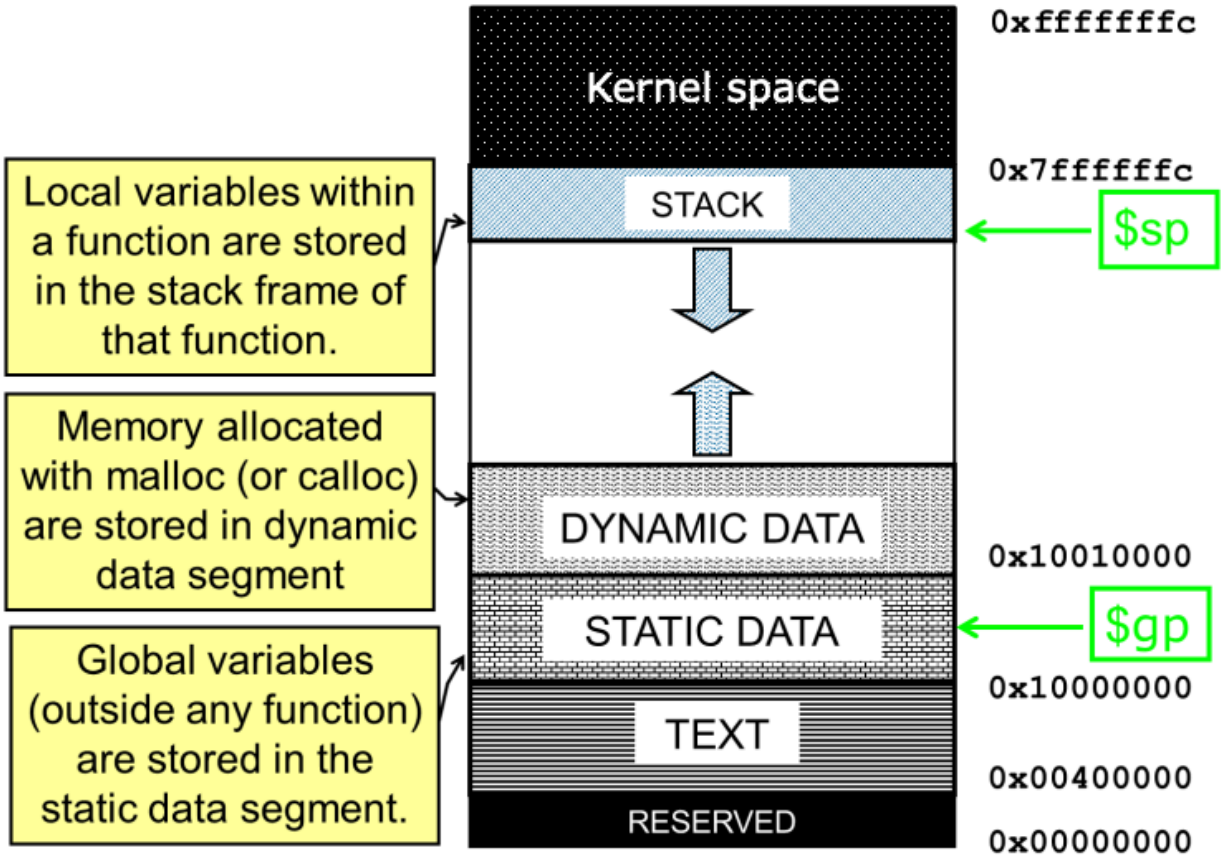

ssy

MIPS has two coprocessors, in addition to the CPU:

**Coprocessor 0** - used to handle exceptions (more info in system calls)
**Coprocessor 1** - used to store/perform arithmetic operations with floating point units (more in basic data types)

# System Calls

**Summary of memory regions:**




Stack pointer - always points to the smallest address allocated on stack
Global pointer - points to middle of static data segment - allow efficient access in the static data segment via base displacement method

Kernel space take up half of all memory

## Kernel

- Handles all exceptions e.g. overfllow exception
- When an exception occurs, PC changes to address stored in kernel space
  - can store the state to return to program if needed
  - can run instructions (i.e. memory addresses) that cannot be accessed in user space

## Exception Handling

- Exception code is loaded to coprocessor 0's `cause` register when there is a exception thrown
- Current PC value loaded to coprocessor 0's `epc` register (so kernel can return to that instruction (in user code) after handling exception- helps to know what value of PC to change to)

- Program counter value gets changed to `0x8000000` (location of exception handling code - fixed address in kernel space) - you don't need to write exception code handling (kernel will do it for you)
- $k0 and $k1 may be scratch registers without being restored

## Syscall

- short for **System Call**
- Native CPU instruction with encoding `0x0000000c`
  - R-type instruction (with opcode = rs = rd = rt = 0, funct number $c_{16}$)
- Can possibly trigger a syscall exception (exception number 8)
- allows to access a set of OS systems
- register $v0 indicates the syscall number
- Otherwise follows convention of taking input parameters in $a0 - $a3 (depending on hwo many arguments needed by the syscall)
- example of services accessed via syscall: printing ints/floats/doubles/strings, reading(input) ints/floats/doubles/strings/characters, exit program

Syscall table

| Service | System Call (loaded into $v0) | Arguments | Results |
|---|---|---|---|
| print_int | 1 | $a0 = integer | |
| print_float | 2 | $f12 = float | |
| print_double | 3 | $f12 = double | |
| print_string | 4 | $a0 = string | |
| read_int | 5 | | integer (in $v0) |
| read_float | 6 | | float (in $f0) |
| read_double | 7 | | double (in $f0) |
| read_string | 8 | $a0 = buffer<br>$a1 = length | |
| exit | 10 | | |
| read_char | 12 | | char (in $v0) |

example - reaidng 2 numbers and writing their sum

```
# Interactive calculator - add 2 numbers together

        .data
str1:  .asciiz "\nEnter first value: " # Note initial newline.
str2:  .asciiz "Enter second value: "
str3:  .asciiz "Sum is equal to "


        .text
        .globl main
main:  addi $v0, $0, 4      # Set system call code (print_string)
        la   $a0, str1      # Load address of string into $a0.
        syscall             # Print string.

        addi $v0, $0, 5     # Set system call code (read_int)
        syscall             # Read integer into $v0.
        add $t1, $0, $v0    # Store integer in temporary register.
```

result of integer input is stored in register $v0, so store it in $t1 temporarily - as we need to do another syscall so have to modify the value in $v0 (as syscall codes are stored in v0)

```
    addi $v0, $0, 4     # Set system call code (print_string)
    la   $a0, str2      # Load address of string into $a0.
    syscall             # Print string.

    addi $v0, $0, 5     # Set system call code (read_int)
    syscall             # Read integer into $v0.
    add $t1, $t1, $v0   # Add newly entered value to previous.

    addi $v0, $0, 4     # Set system call code (print_string)
    la   $a0, str3      # Load address of string into $a0.
    syscall             # Print string.

    addi $v0, $0, 1     # Set system call code (print_int)
    add  $a0, $0, $t1   # Print result.
    syscall             # Call the system service.

    li $v0 10           # Exit program using system call 10.
    syscall             # syscall 10 (exit)
```

note that `addi $v0 $0 $5` sets v0 to 5 (it doesn't add)

# Basic Data Types

## Representing Characters

**ASCII (7 bytes)**

| ASCII value | Character | ASCII value | Character | ASCII value | Character | ASCII value | Character | ASCII value | Character | ASCII value | Character |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | space | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | $ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | ( | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 | ) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | \| |
| 45 | - | 61 | = | 77 | M | 93 | ] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | DEL |

3 most signifcant bits = columns, 4 least significant bits = the rows

you perform a **bitwise and** on not `0b100000` to ensure you return the capitalised letter (bit masking)

to swtich the case of a letter, you flip the MSB bit (perform XOR on `0b100000`)

to get a value of a digit you perform `y - '0' = y - 48`

ASCII values 0-31 are non-printable (control) characters

**Extended ASCII (8 bits)**

- represent characters with 8 bits (bytes)
- can represent other characteers not in ASCII (e.g. the pound symbol)

**Unicode**

- Uses up to 4 bytes per character
- Can represent all the world characters and emojis (around 145k characters)
- Has muiltiple encodings e.g. UTF-32 (uses 32 characters) - but brings up endianness (resolved by special character that precedes the text)

**UTF-8**

- Efficient, backward compatible Unicode encoding that is widely used today
- Variable-length encoding
- ASCII characters are **only** represented as 1-byte characters (they start with 0 in UTF-8, followed by the bits)

- 2 bytes encoding - first byte starts with 110, second byte starts with 10 - can represent 2048 different characters
  - e.g. the british pound
  - covers latin-script/greek/cyrillic/hebrew/arabic characters
- 3 bytes encoding - first byte start with 1110, second byte start with 10, third byte starts with 10
- 4 bytes encoding - first byte start with 11110, second byte 10, third byte 10 and last byte start with 10
- number of bytes needed to encode depends on the number of 1s in the first byte
- can't use all 32 bits to represent different characters as UTF-8 parser needs to know how many bytes does the character encoding have

**Characters to String**

- in C, the last character of string is the null character - tells where the string ends
- Python 'str' and Java 'String' objects are abstractions of Unicode characters

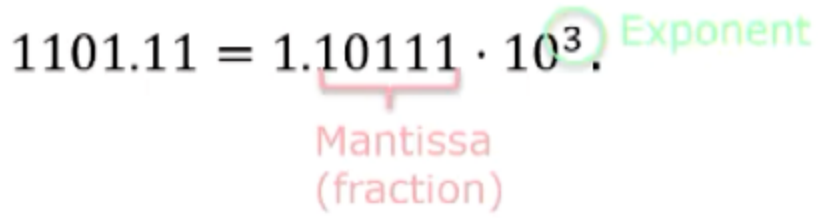
# Floating Point Representation

## Single Precision

- Used if we don't know the scale, so it has to be reprsented as part of the number
- Only fixed amount of mantissa bits are encoded (so there is accuracy loss)
- exponent is not encoded directly but with a bias
  - exponent argument is in range `{1, ... 254}` in natural numbers
  - it gets adjusted where the true exponent is equal to EA - B (where B is $127_{10}$ = bias)
  - true exponent is in range $[1 - 127, 254 - 127] = [-126, 127]$
  - allows to represent numbers that are both smaller/larger than 1 (both positive/negative)
  - most significant bit of exponent argument can indicate if exponent is positive or negative (msb = 1 => positive exponent, otherwise is negative exponent)
- value represented = $(-1)^{\text{sign}} \times 2^{EA-B} \times 1 \times \text{fraction}$
- single-precision floats represented in 32 bits:


IEEE 754 format for single-precision floats

e.g. (difference between mantissa/exponent)




representation example

- E.g., 11000001101000000000000000000000

$$\text{represents } -2^{131-127} \cdot 1.01_2 = -16 \cdot 1.25 = -20_{10}.$$

Largest number represented: $(2 - 2^{-23}) \times 2^{127} \approx 2^{128}$
Smallest normal number represented: $2^{-126}$

**Subnormal numbers**

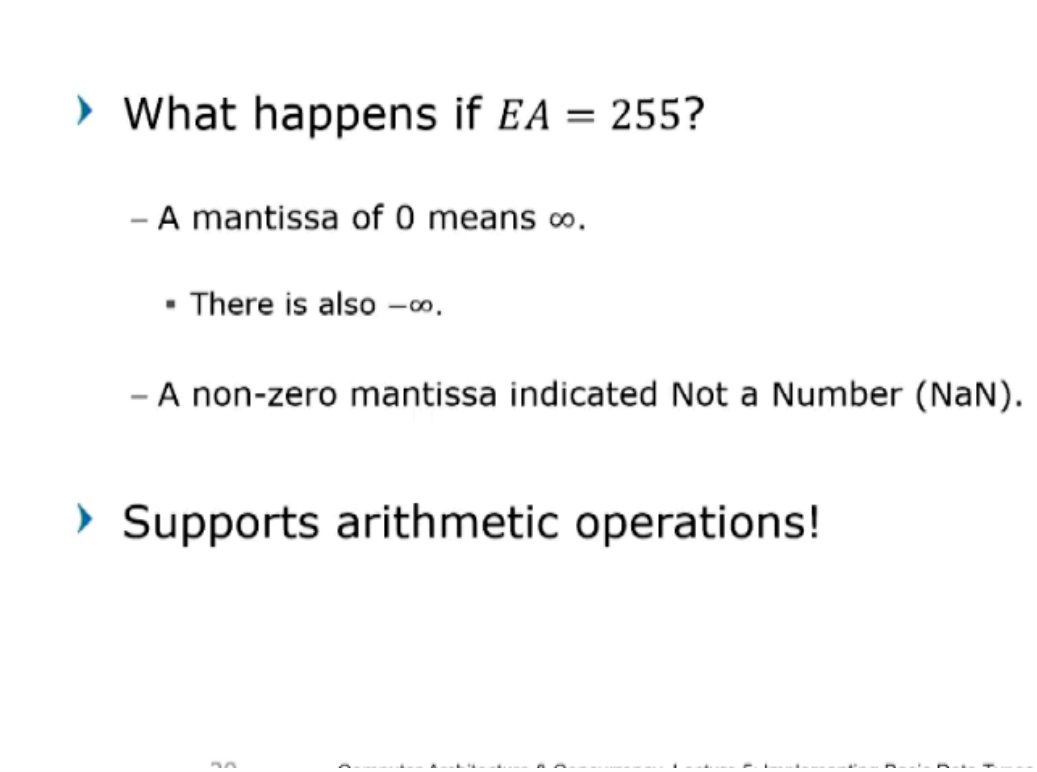
- Represented with EA of 0
- Represents $(0 \times \text{fraction} \times 2^{-126})$
- smallest subnormal number: $-2^{-128}$
- largest subnormal numbers (set all mantissa bits to 1): $2^{-149}$
  - 10000000001000000000000000000000 represents $-2^{-126} \cdot 0.01_2 = -2^{-128}$.
  - 00000000011111111111111111111111 represents

    $$2^{-126} \cdot 0.1\ldots1_2 = 2^{-126} \cdot (1 - 2^{-23}).$$
  - 00000000000000000000000000000001 is $+2^{-126} \cdot 0.0\ldots01_2 = 2^{-149}$.
  - 00000000000000000000000000000000 is +0 (and there's also -0!).

  yuo can check the sign of a number even if a number is very close to 0

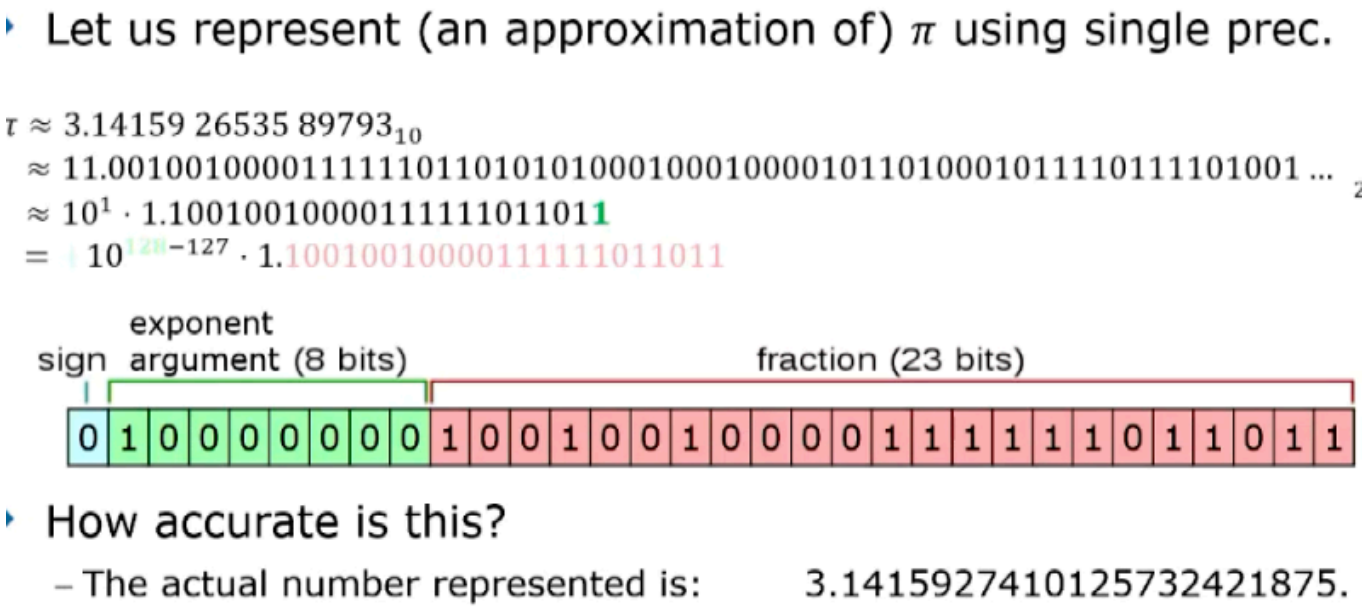
**Special Numbers**

- When EA = 255 (can be either NaN or $\pm\infty$)




| Operation | Result |
|---|---|
| n ÷ ±Infinity | 0 |
| ±Infinity × ±Infinity | ±Infinity |
| ±nonZero ÷ ±0 | ±Infinity |
| ±finite × ±Infinity | ±Infinity |
| Infinity + Infinity<br>Infinity − -Infinity | +Infinity |
| -Infinity − Infinity<br>-Infinity + − Infinity | − Infinity |
| ±0 ÷ ±0 | NaN |
| ±Infinity ÷ ±Infinity | NaN |
| ±Infinity × 0 | NaN |
| NaN == NaN | False |

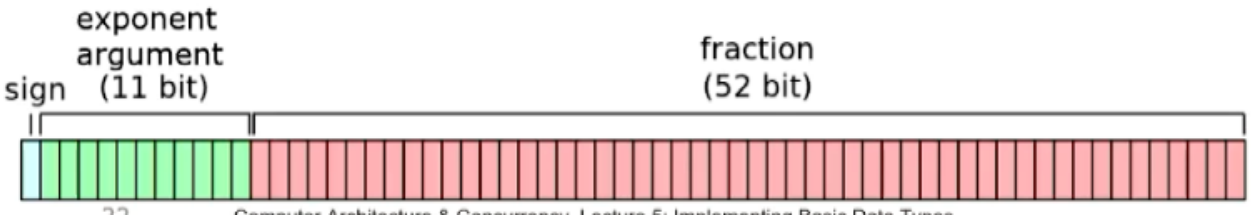
example - representing $\pi$ with single precision




## Double Precision

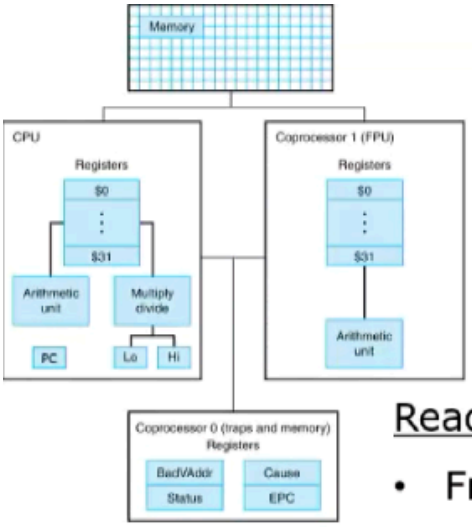
- similar to single precision but with 64 bits
- bias B = 1023
- normal numbers have exponent argument $EA \in [1, 2046]$
- EA = 0 used for subnormals

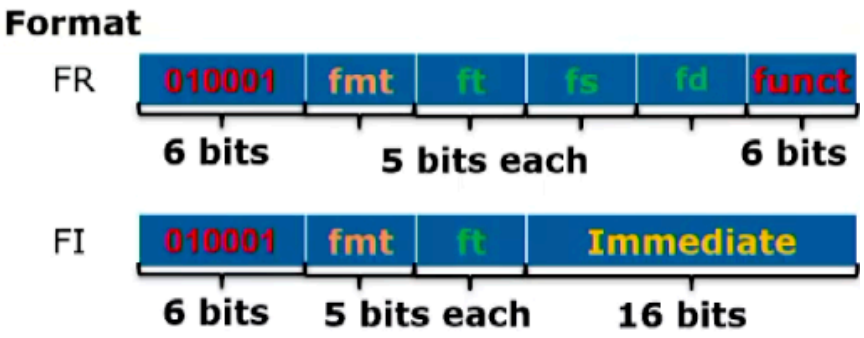
- EA = 2047 for NaN and $\pm\infty$




## Representing floating points in MIPS

Via coprocessor 0




Reading to FP registers:

- From memory:

```
lwc1 $f1, 100($s2) Load word to cop. 1.
swc1 $f1, 100($s2) Store word from cop. 1.
```

- From Registers:

```
mtc1 $t2, $f2 Load word to cop. 1.
```

note that floating point units can't be stored in registers but in coproecessor 0 (floating point registers)

**Arithmetic**

| Instruction | Description |
|---|---|
| `add.s $f2, $f4, $f6` | FP add (single precision) |
| `sub.s $f2, $f4, $f6` | FP subtract (single precision) |
| `mul.s $f2, $f4, $f6` | FP multiple (single precision) |
| `div.s $f2, $f4, $f6` | FP divide (single precision) |
| `add.d $f2, $f4, $f6` | FP add (double precision) |
| `sub.d $f2, $f4, $f6` | FP subtract (double precision) |
| `mul.d $f2, $f4, $f6` | FP multiple (double precision) |
| `div.d $f2, $f4, $f6` | FP divide (double precision) |

**Converting floats to ints**

Use the `cvt` (convert) instructions.

- `cvt.s.w` (convert word to single-prec. float)
- `cvt.d.w` (convert word to double-prec. float)
- `cvt.d.s / cvt.s.d` (convert single to double or vice versa)
- `cvt.w.s / cvt.w.d` (convert to word)

only MIPS64 can only convert to long (irrelevant but fun fact)