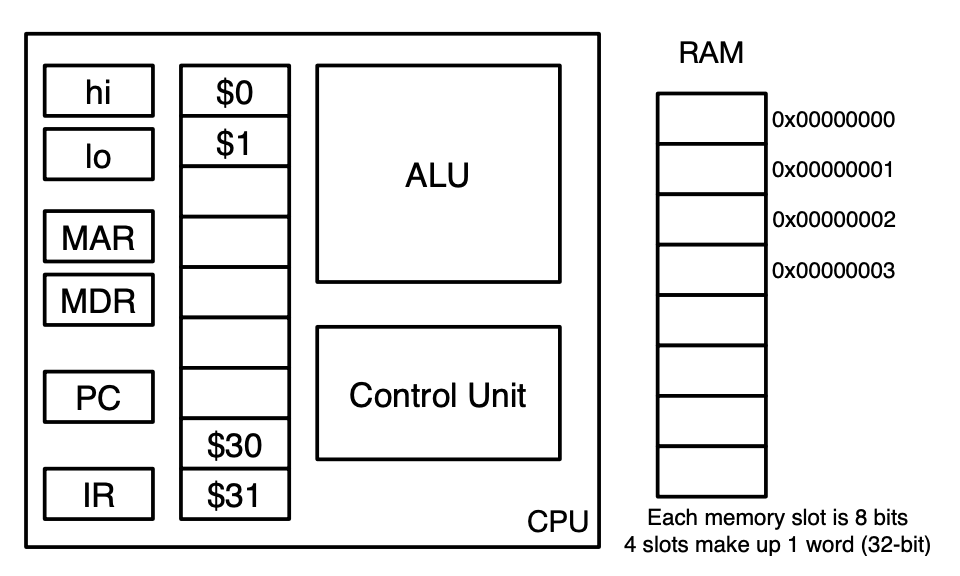
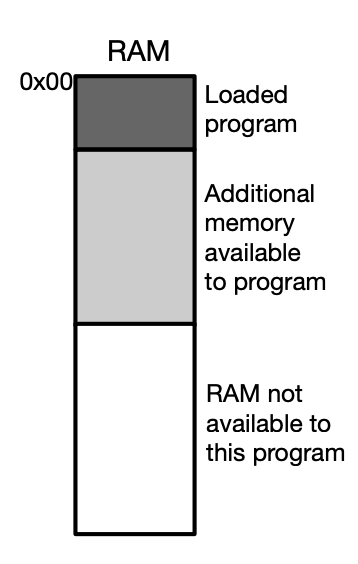
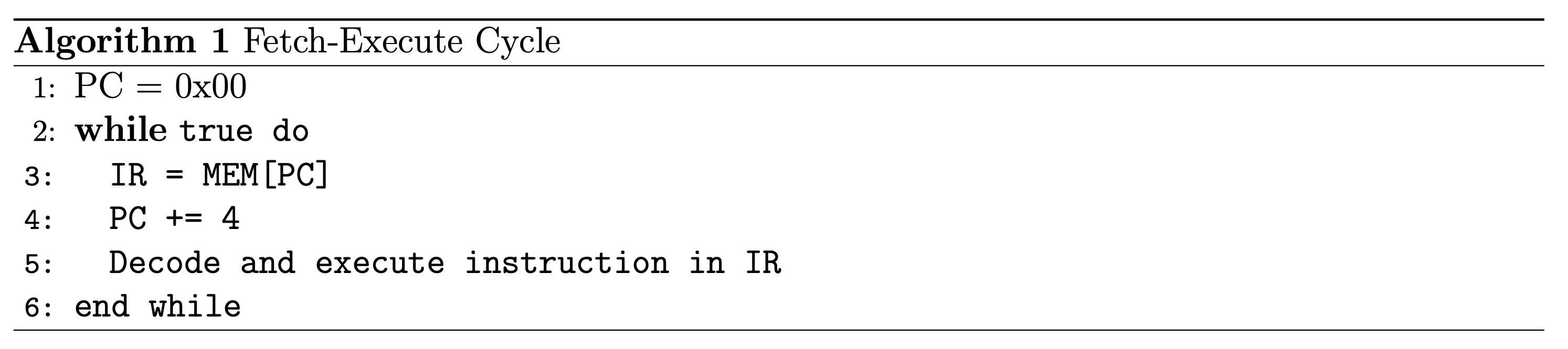
MIPS Hardware

* different processors understand different patterns (i.e. machine language)
  + a machine language is made up of multiple patterns and each is called a machine instruction
  + many different processor-specific machine languages exist
* in this course, we will use machine language called 32-bit MIPS (Microprocessor Without Interlocked Pipeline Stages)
* below is simplified MIPS processor and RAM (Random Access Memory)



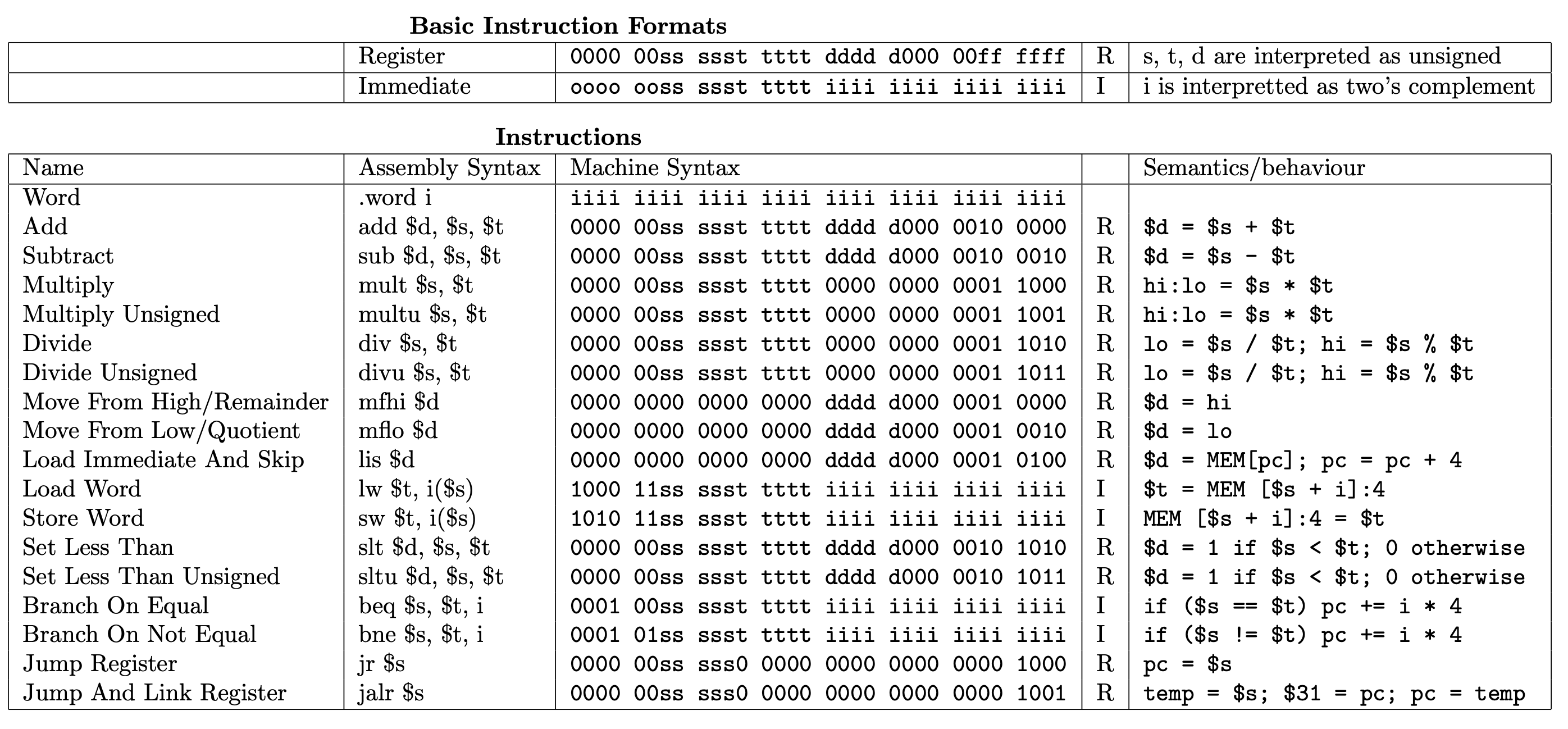
* + Control Unit is brain of processor
    - controls sequence of instructions to execute, interprets the instructions and flow of data, contains the processor’s timing unit, and controls signals to and from peripheral devices
  + Arithmetic Logic Unit (ALU) performs mathematical operations such as addition, multiplication, logical comparison, and decisions requested by control unit
  + general purpose registers each hold 32 bits
    - labelled from $0 to $31
    - not actually general purpose and some have special uses dictated by MIPS
      * $0 is always 0
      * $31 is for return addresses
      * $30 is stack pointer
      * $29 is frame pointer
  + Program Counter (PC) and Instruction Register (IR) are 2 special 32-bit registers used by the Control Unit in fetching and decoding instructions
  + Memory Data Register (MDR) and Memory Address Register (MAR) are used when data travels through data bus
    - when CU wants to read from memory address, it’s stored in the MAR
    - data fetched from address is placed into MDR and then loaded into appropriate register
  + hi and lo registers are used by multiplication and division machine instructions
  + off-chip memory is RAM
    - array of 8-bit slots
    - 4 RAM addresses comprise a single word so we use hexadecimal rep for memory addresses
* to run a program, the program (sitting on hard drive) has to be loaded into memory
  + OS uses special Loader program to do this and we assume it’s loaded at start of memory (i.e. address 0x00)
  + once a program is loaded, memory looks like:



* + loader must do other things to set up other sections of memory for program
    - e.g. loader communicates to processor where it’s loaded the program by setting the Program Counter (PC) register to starting address (0x00 in this case)
* once the loader is done, a hardware-based algorithm called the Fetch-Execute Cycle starts running
  + below is pseudocode of the algorithm
* 
  + - infinite loop unless instruction is executed that breaks out of it
    - line 3: reads the machine instruction that starts at memory address PC and stores it into Instruction Register (IR)
    - line 4: all registers and machine instructions are 32 bits and each address of memory is 8 bits, there must be increment of 4 bytes to PC so next instruction can be received
  + this is a program that runs other programs
  + order is important and PC is updated before instruction is actually executed so may affect meaning of instructions that interact with PC

MIPS Machine Language

* below is simplified set of 18 instructions in MIPS machine language:



* add instruction takes values stored in 2 registers (we’ll call them $s and $t) and stores the result in another register $d (i.e. $d = $s + $t)
  + need 5 bits to rep register number since there’s 32 general purpose registers
  + add instruction template is:



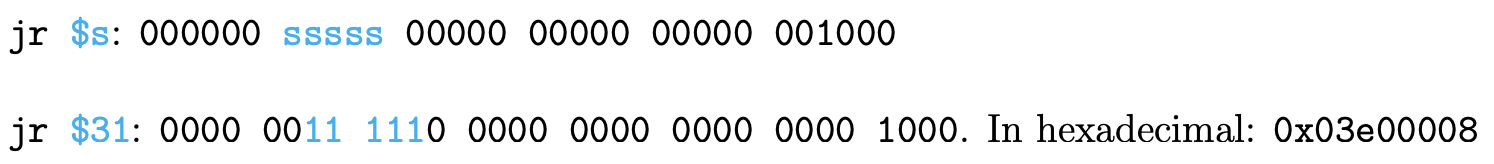
* + another format is below, to make relationship with hexadecimal clearer



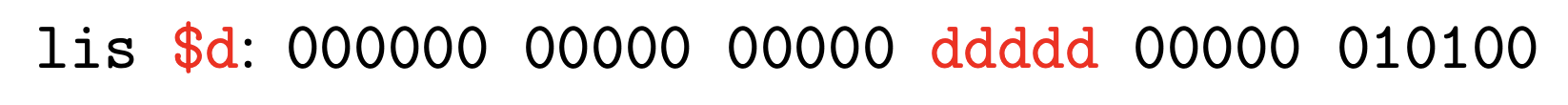
* + note difference in order that we specify the registers in assembly syntax vs machine syntax
* e.g. write a program in MIPS that takes in the values of registers $5 and $7, adds them together, then stores the result in register $3
  + solution is:



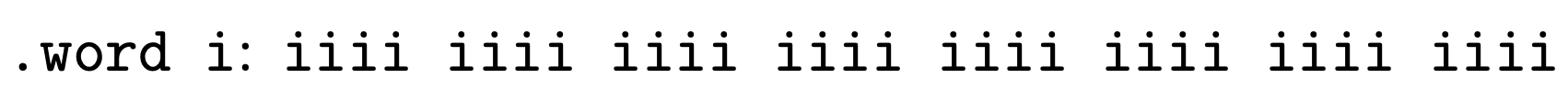
* to negate value, simply subtract it from 0: sub $3, $0, $3
* jump instruction in MIPS changes control flow in program
  + changes PC to value in jump register $s
  + used to exit MIPS program: jr $31
    - $31 is register for return address
    - unlike most HLL where program exits when last instruction is executed, MIPS programs are expected to have an explicit jump to the return address stored by the loader



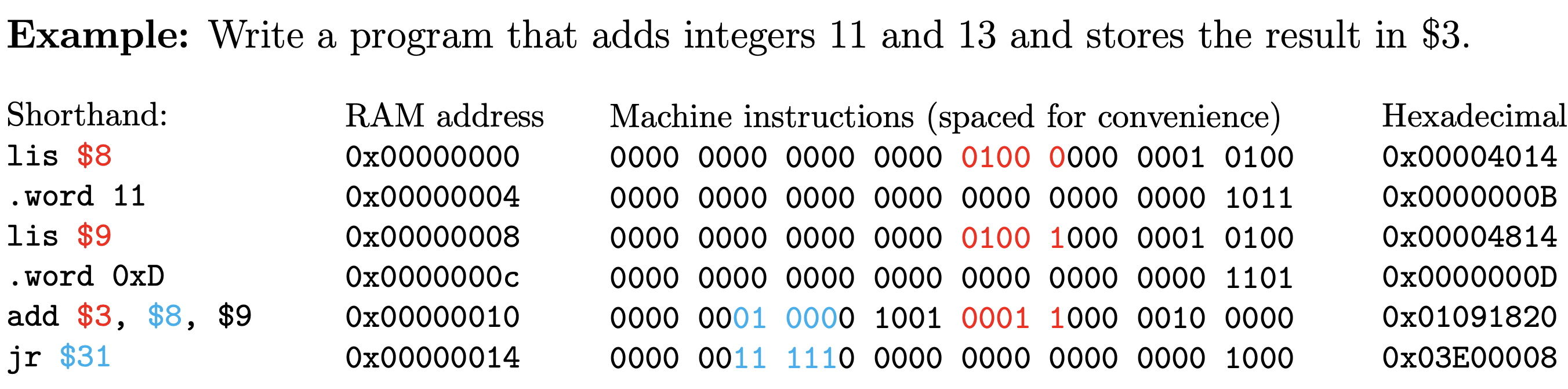
* bits in registers can be interpreted differently
  + e.g. add and sub instructions interpret them as signed integers but jr instruction interprets them as an address
* to put values (constants called immediates) in registers:



* + stands for load immediate and skip
  + places the next value in RAM into $d and increments the program counter by 4 so that it skips the next line, which is usually not an instruction



* + this places the immediate value we want into the register
* below is a complete example of a MIPS program



MIPS Assembly Language

* assembly language is a textual rep of a machine language
* multiplying two words together might give a word that requires twice as much space
  + to fix this, use registers hi and lo
* mult $s, $t performs multiplication and places the most significant word (4 bytes) in hi and least significant word in lo
  + mult assumes 2 numbers are in signed two’s complement rep but there’s a multu instruction for unsigned ints
  + unless explicitly stated, assume result of multiplying 2 registers is 32-bit value that fits into lo register and doesn’t overflow into hi register

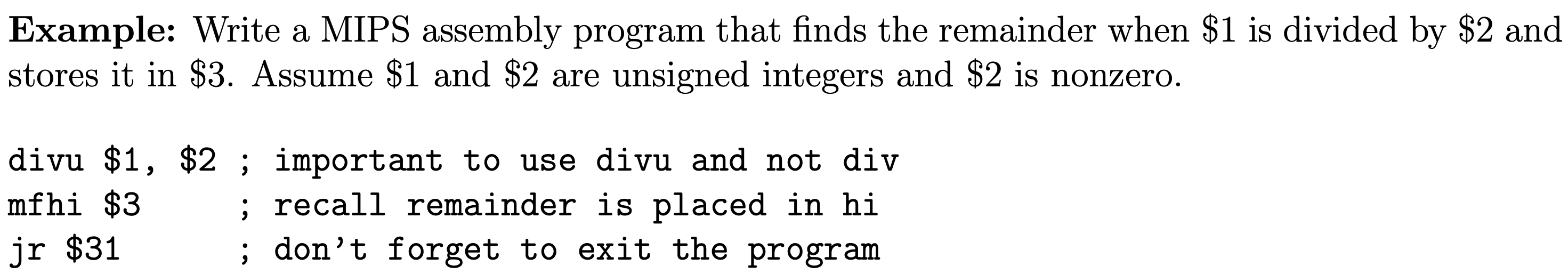


* div $s, $t performs integer division and places the quotient $s / $t in lo and remainder $s % $t in hi
  + sign of remainder matches the sign of the dividend stored in $s
  + divu instructions for unsigned ints



* since results of mult and div are stored in hi and lo registers, there’s special instructions to access the data

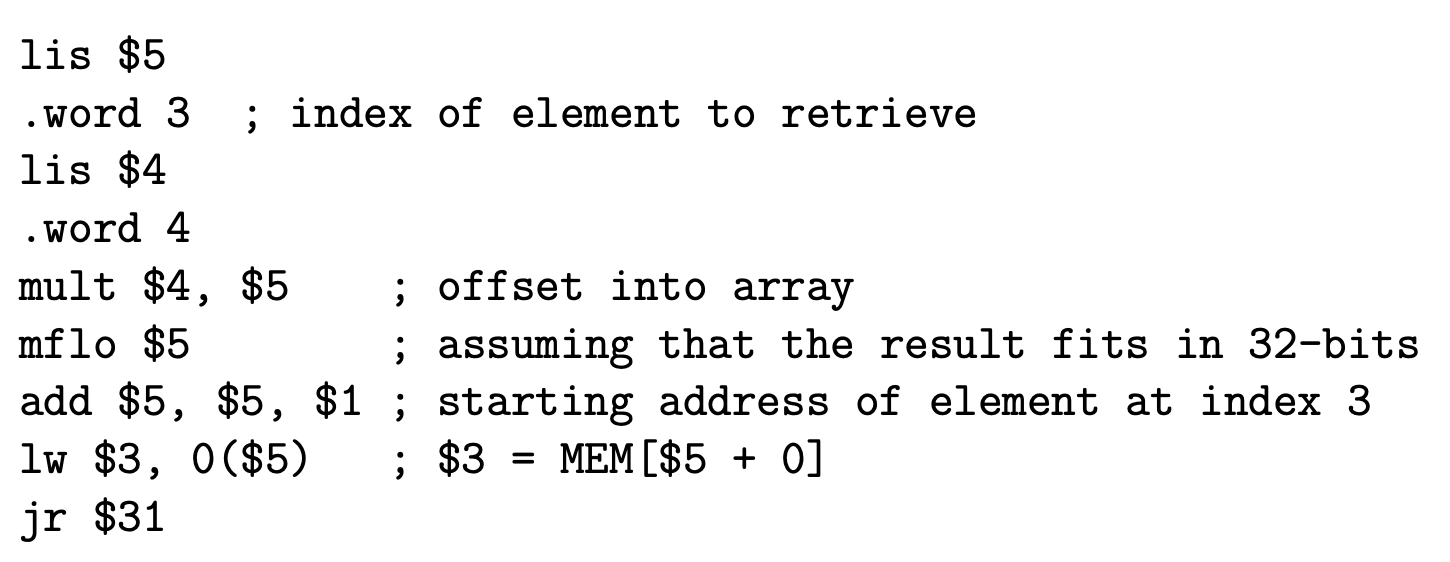
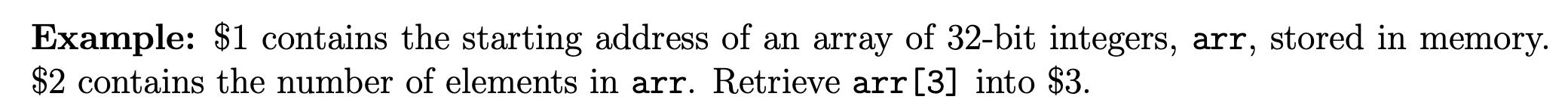




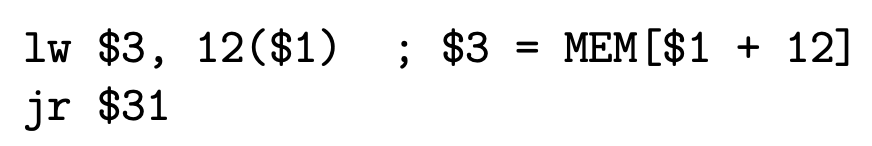
* larger amount of memory called Random Access Memory (RAM) is stored off the CPU
  + RAM access is slower than register access but it’s larger
  + data travels between RAM and CPU by bus
  + instructions in RAM start with address 0 and increase by word size (4 bytes)
* each memory block in RAM has an address (one address is one byte)
  + words occur every 4 bytes and are formed from consecutive, (usually) aligned bytes
  + can’t use data directly from RAM, must load into register first
* load word: lw $t, i($s) takes a word from RAM and places it into a register
  + load the word in MEM[$s + i] and store in $t



* store word: sw $t, i($s) takes a word from a register and stores it into RAM
  + load the word in $t and store it in MEM[$s + i]
* the i must be immediate and not another register



Simpler solution:

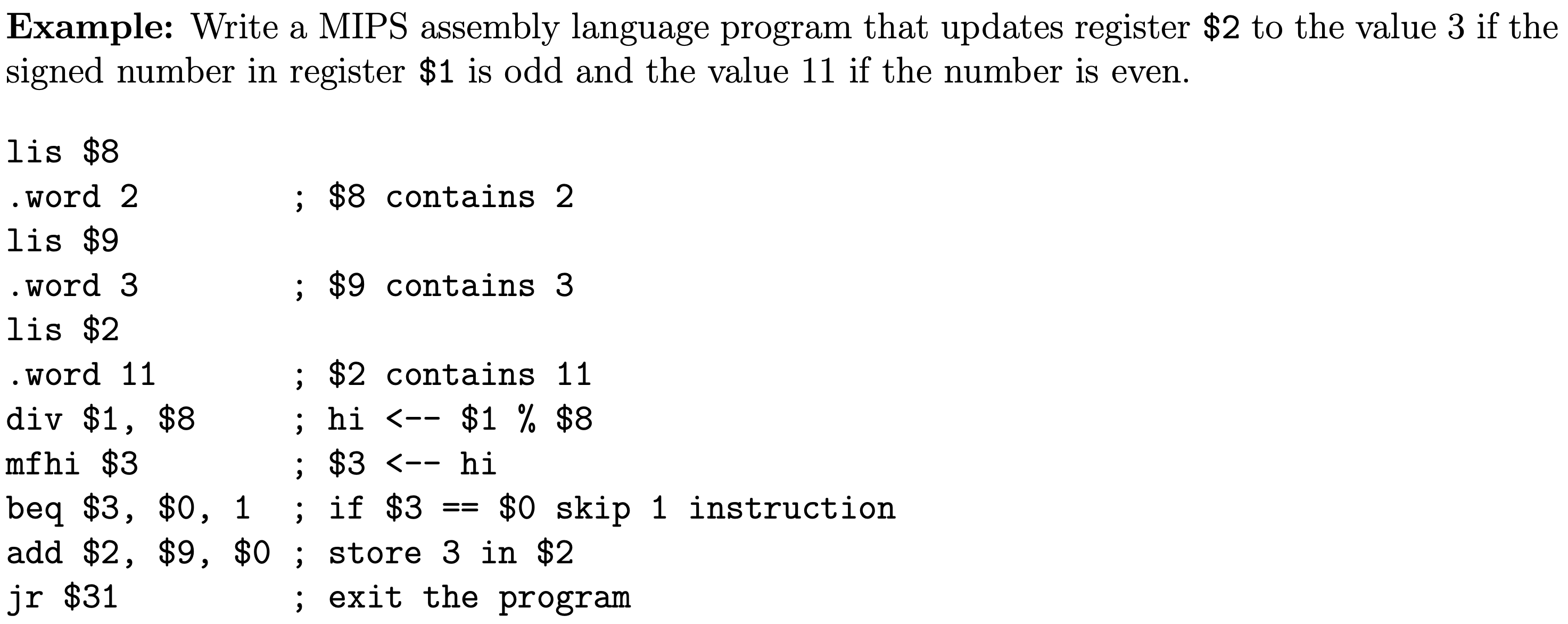


* MIPS also has control statements
  + beq $s, $t, i means branch on equal
    - if $s == $t, then pc += i \* 4
    - i.e. skip ahead i many instructions if $s and $t are equal



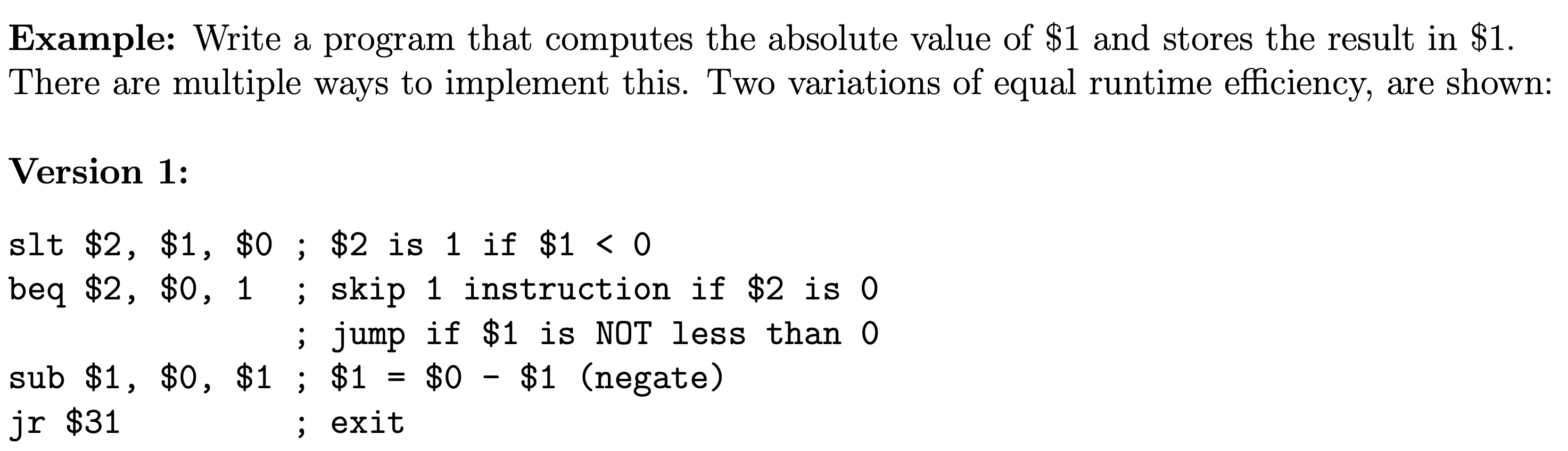
* + bne $s, $t, i means branch on not equal
    - if $s != $t, then pc += i \* 4
    - i.e. skip ahead i many instructions if $s and $t are not equal

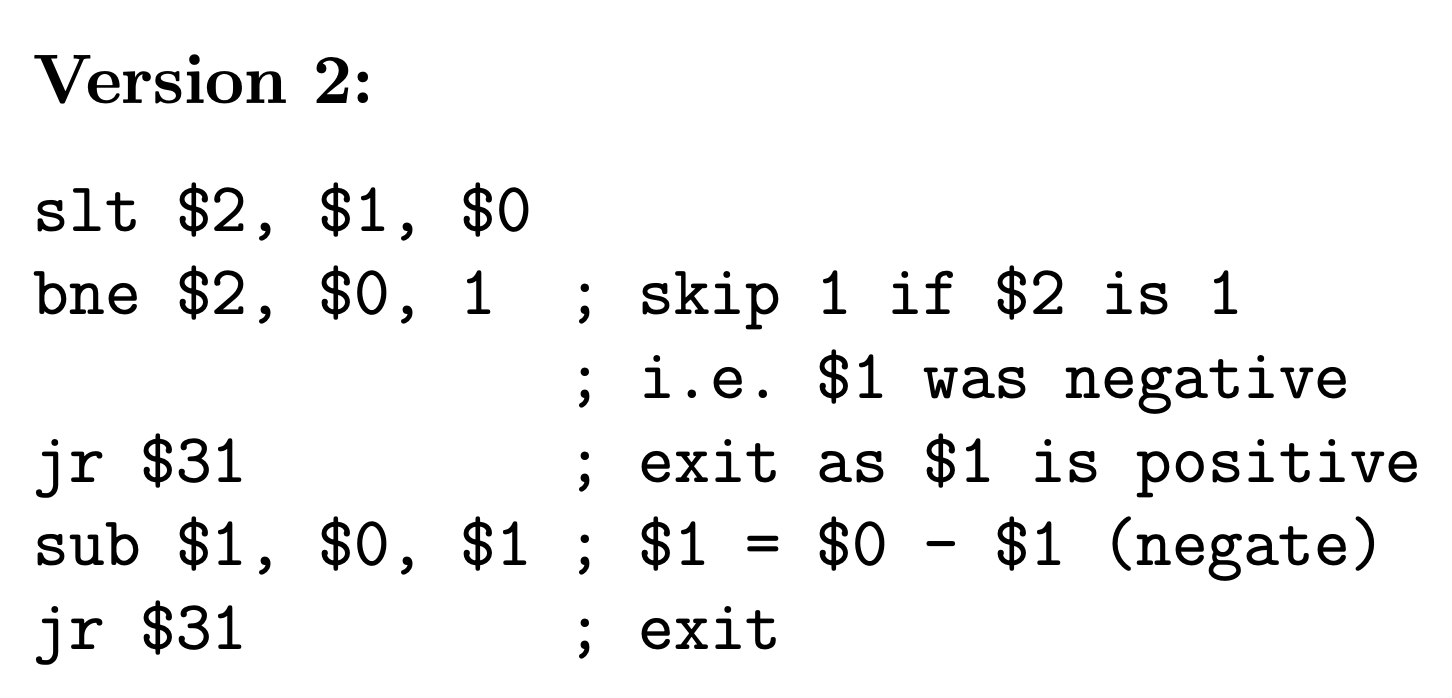




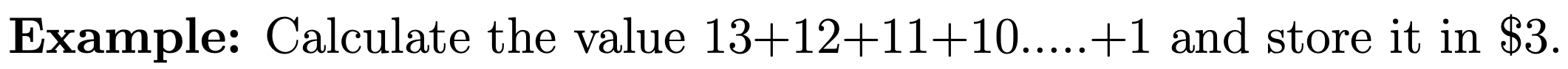
* slt $d, $s, $t means set less than
  + sets value of register $d to be 1 provided the value in register $s is less than the value in register $t and sets it to be 0 otherwise
  + there’s an unsigned version of instruction: sltu

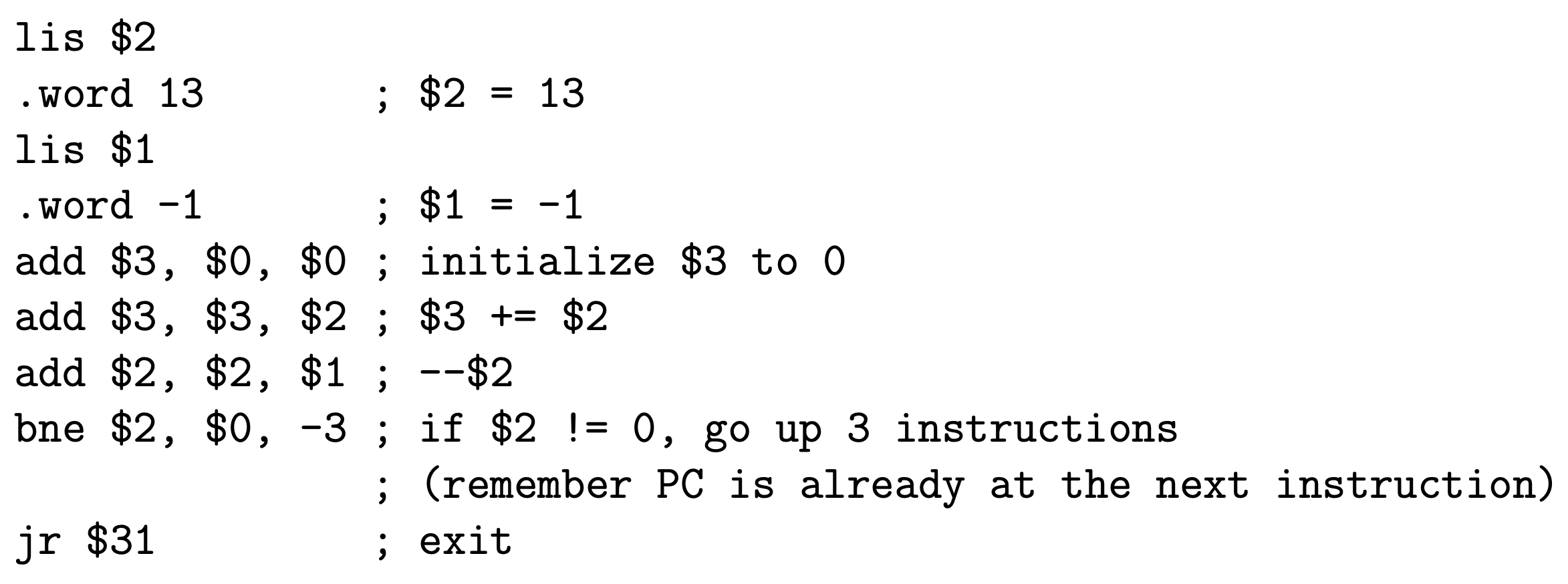




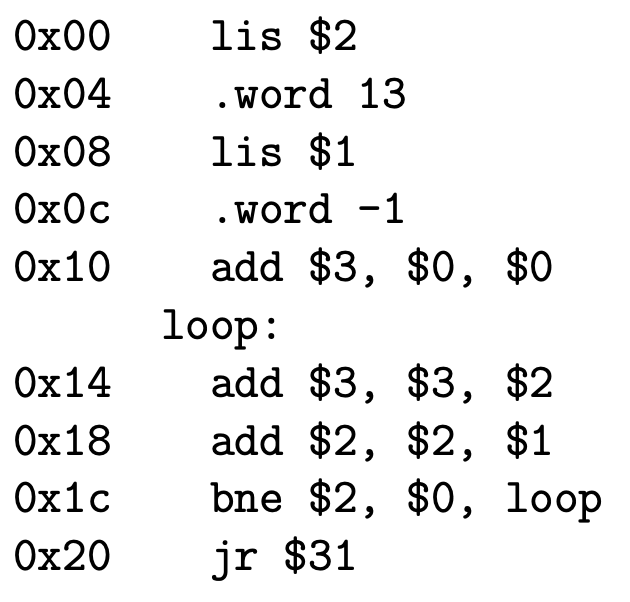


* with branching, we can also do looping since they don’t exist as separate instructions in MIPS assembly language

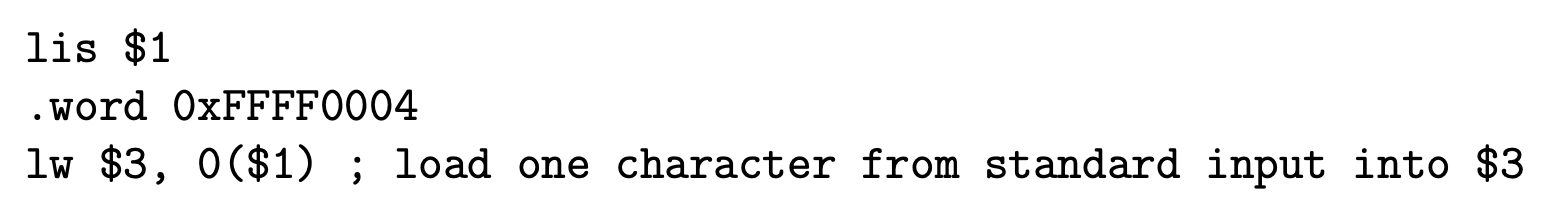




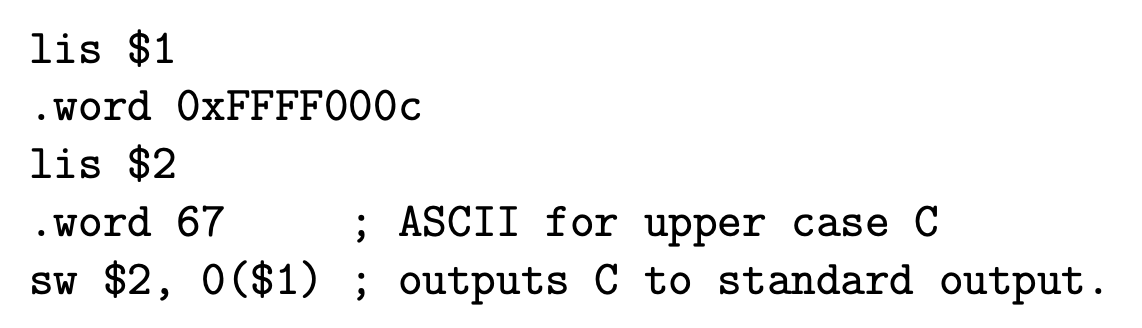
* + hard-coding –3 could lead to future problems in case we add another line of code in between
* labels are assembler directives and aren’t actually machine code
  + they don’t take words so for bne and beq, labels don’t have line numbers on their own
  + a label at the end of code is allowed and it has the address of the first instruction after the program
* below is same loop using labels:



* + loop has address of 0x14 because they don’t take a word of memory
  + loop is computed as (loop – PC) / 4 where PC is line number after the current line
* MIPS programs can read from standard input using a load instruction to load from the specially designated address 0xFFFF0004
  + loading from this address will read one character from standard input and store it as the LSB within the specific register

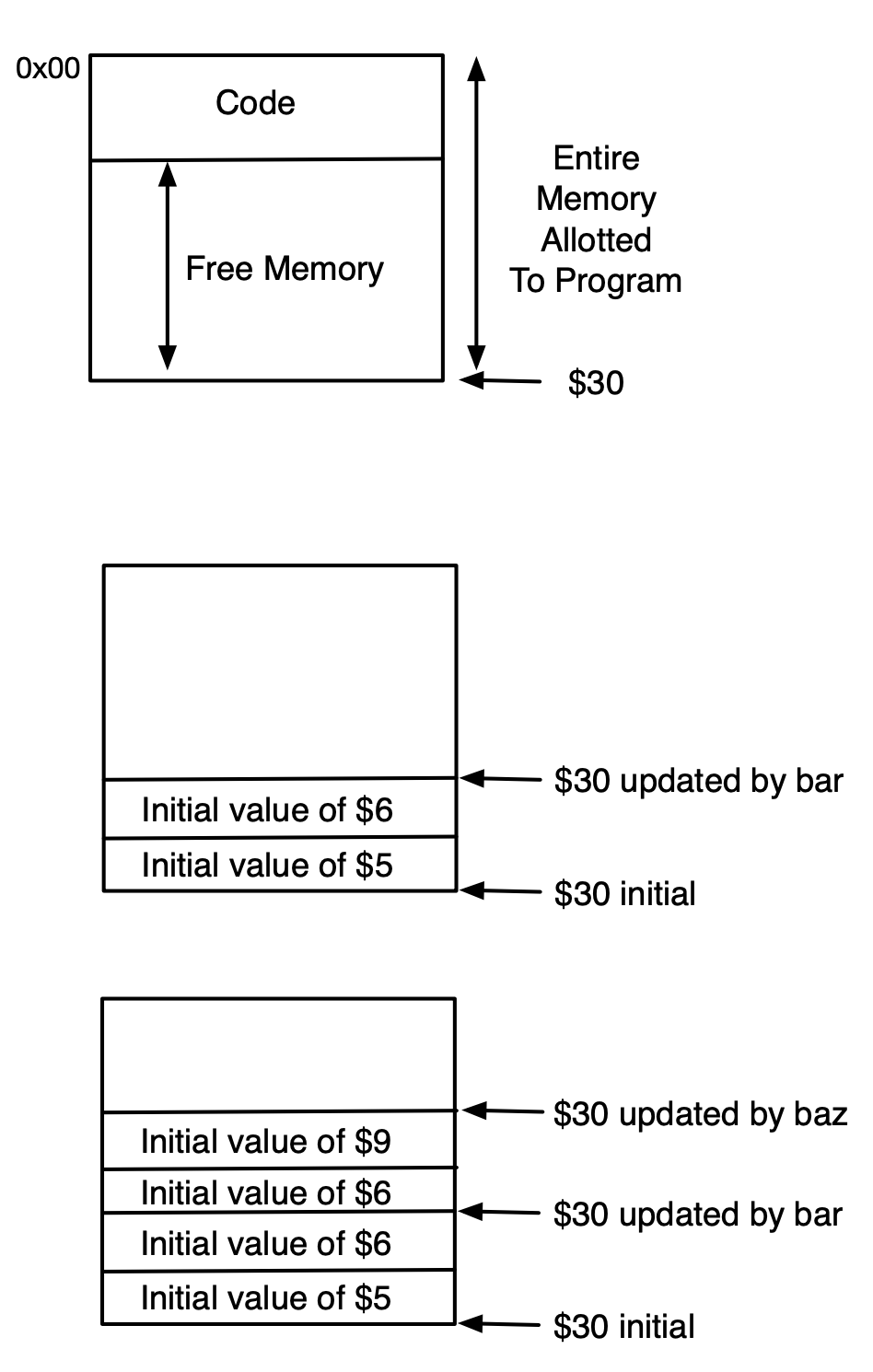


* generate output using a store instruction to write to the specially designated address 0xFFFF000C
  + storing at this address will send LSB in specific register to standard output

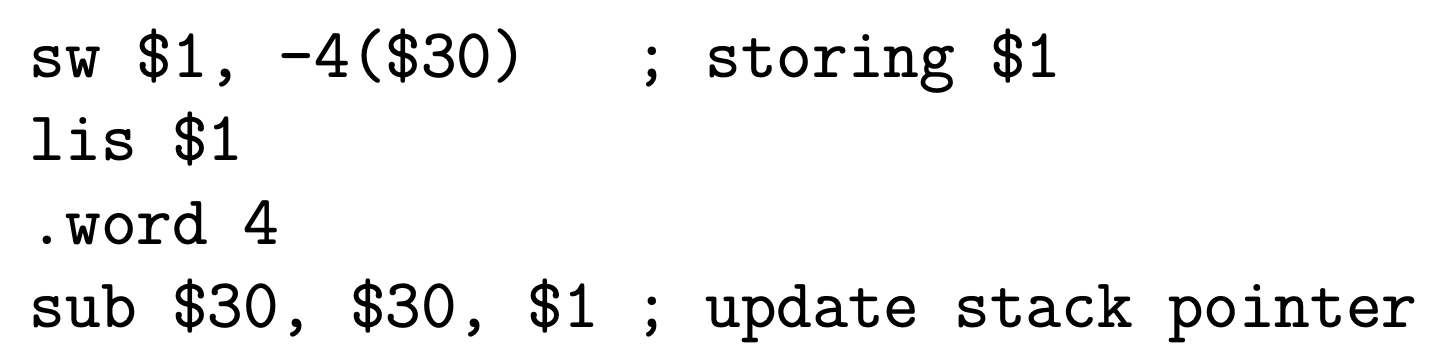


Implementing Procedures In Assembly Language

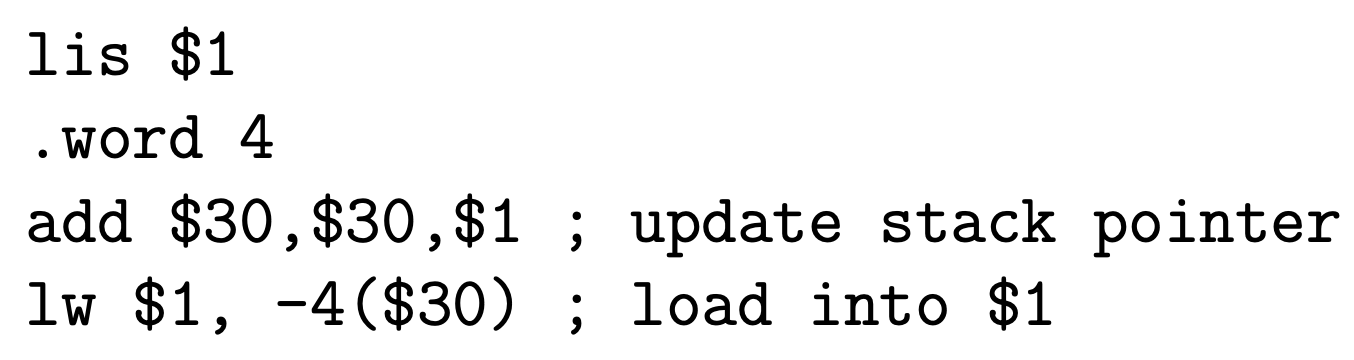
* to protect data in registers, preserve register values by saving and storing them in RAM
* register $30 points to the very bottom of free RAM
  + used as a bookmark to separate the used and unused RAM if we allocate from the one end, and push + pop things like a stack
  + $30 is pointer to the top of a stack
  + stack grows from high memory to low memory
    - push to stack by reducing $30 to get lower addresses
    - pop from stack by



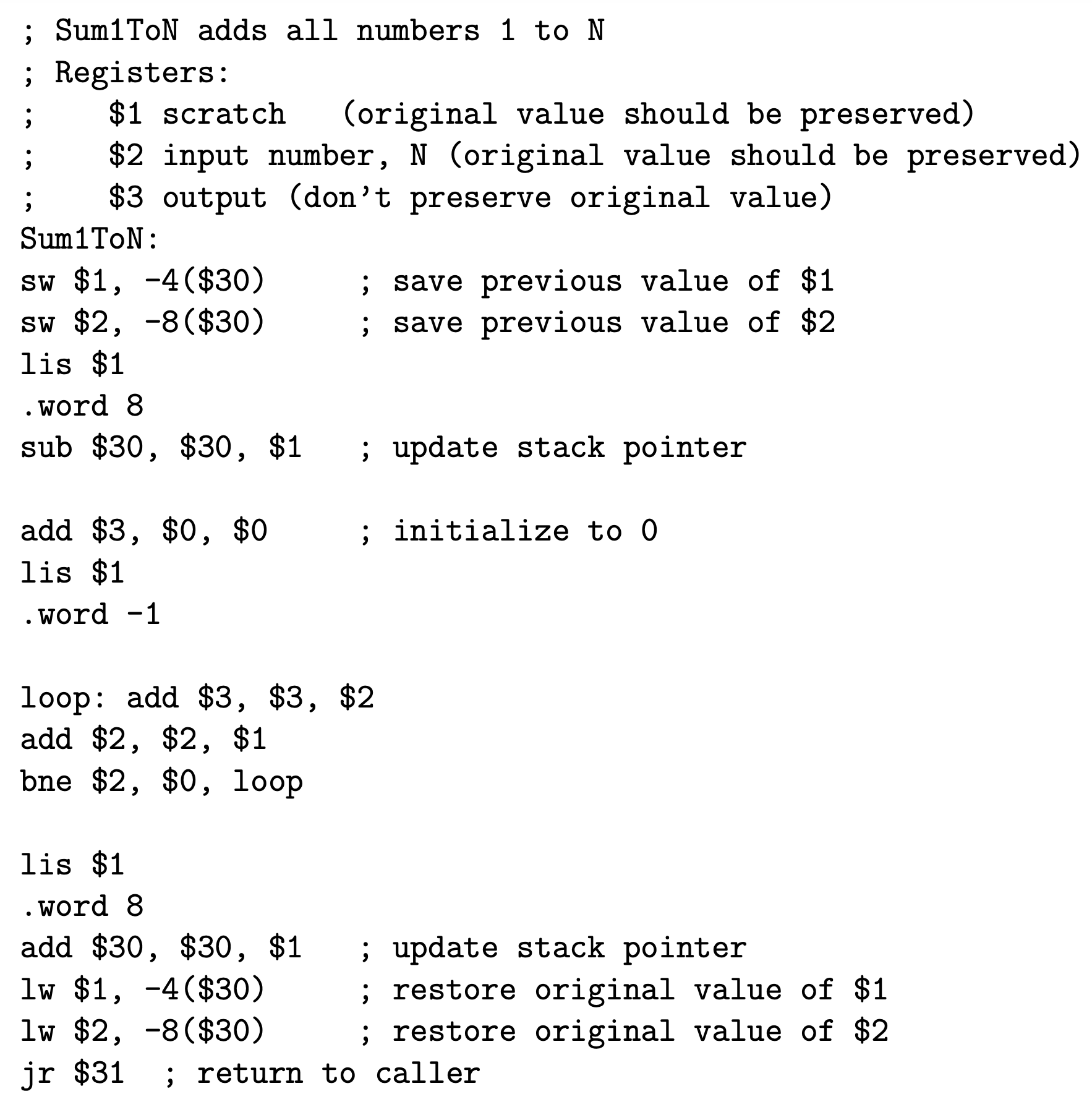
* pushing register value on memory stack:



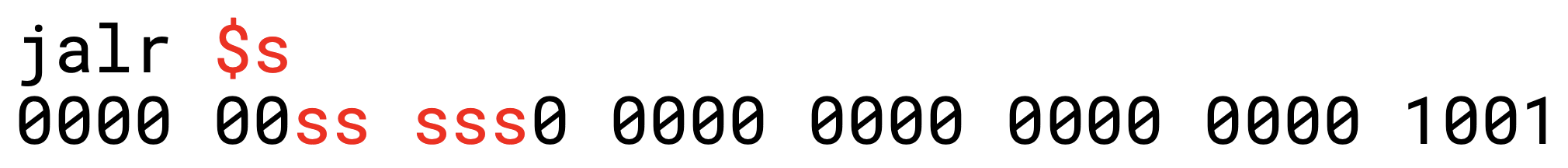
* popping register value off memory stack:



* $30 isn’t stored on the stack but it still gets restored when we reverse the changes we made every time we restore registers from the stack



* + store $1 and $2 in RAM because we need to preserve the values in them before the procedure is run
  + don’t store $3 because the procedure directly modifies the value in it and states that we will place the output value in it
* in order to return to the main program from the procedure, we use jalr which stands for jump and link register



* + sets $31 to be the PC and then sets PC to be $s
  + accomplished by temp = $31, then $31 = PC, then PC = temp
  + since jalr will overwrite $31, first save $31 register to stack
* prologue has all registers that are used by function and stores them in memory stack, body contains all of the function’s code, and epilogue restores all registers from memory stack
  + don’t need to store registers that are being modified by the functions
  + example of calling a procedure from main:

sw $31, -4($30)

lis $31

.word 4

sub $30, $30, $31

lis $1

.word 1234

lis $23

.word SumofDigits

jalr $23

; $2 = 1 + 2 + 3 + 4

lis $25

.word 4

add $30, $30, $25

lw $31, -4($30)

jr $31

; Sums the base ten digits of the

; values of $1. Stores the result in $2.

; Modifies $2

SumofDigits:

; Prologue:

sw $10, -4($30) ;Save $10

sw $3, -8($30) ;Save $3

sw $1, -12($30) ;Save $1

lis $3

.word 12

sub $30, $30, $3

; Body

lis $10

.word 10

add $2, $0, $0

loop:

divu $1, $10

mfhi $3

mflo $1

add $2, $2, $3

bne $1, $0, loop

; Epilogue:

lis $3

.word 12

add $30, $30, $3

lw $10 -4($30)

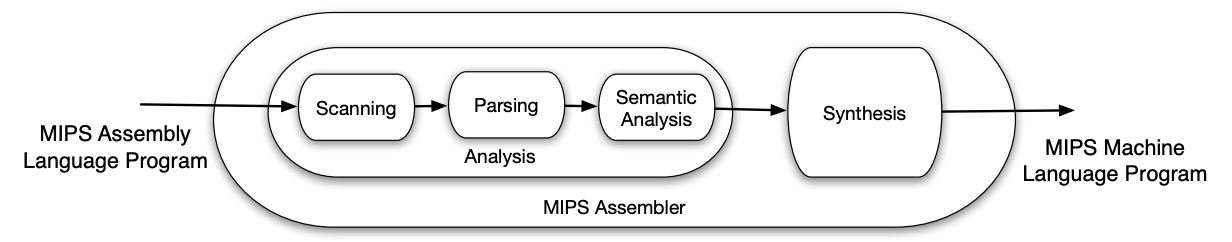
lw $3 -8($30)

lw $1 -12($30)

jr $31

MIPS Assembler

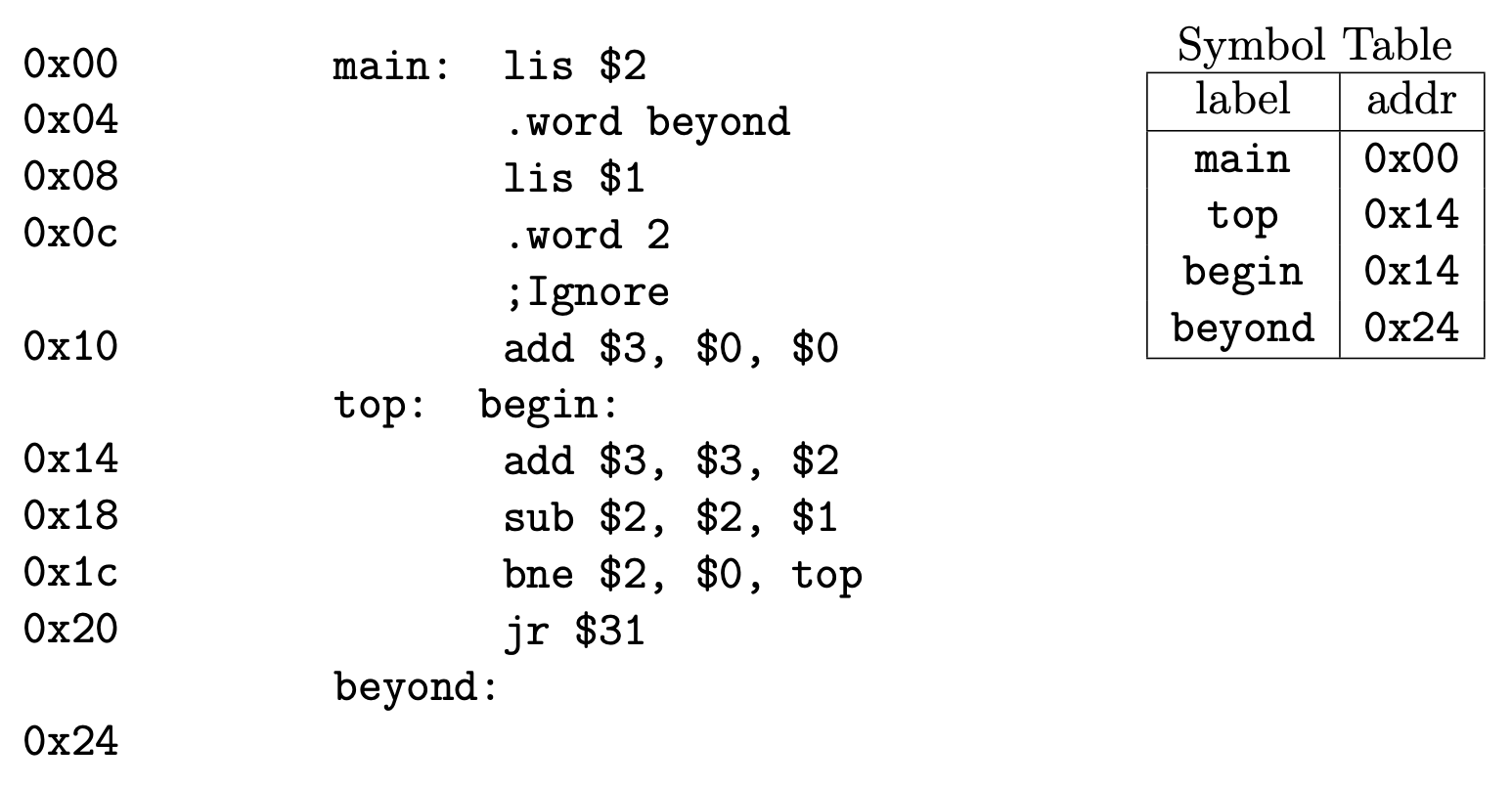
* any translation process involves 2 phases: analysis and synthesis
  + analysis: understand what’s meant by input source
  + synthesis: outputs the equivalent target code in new format



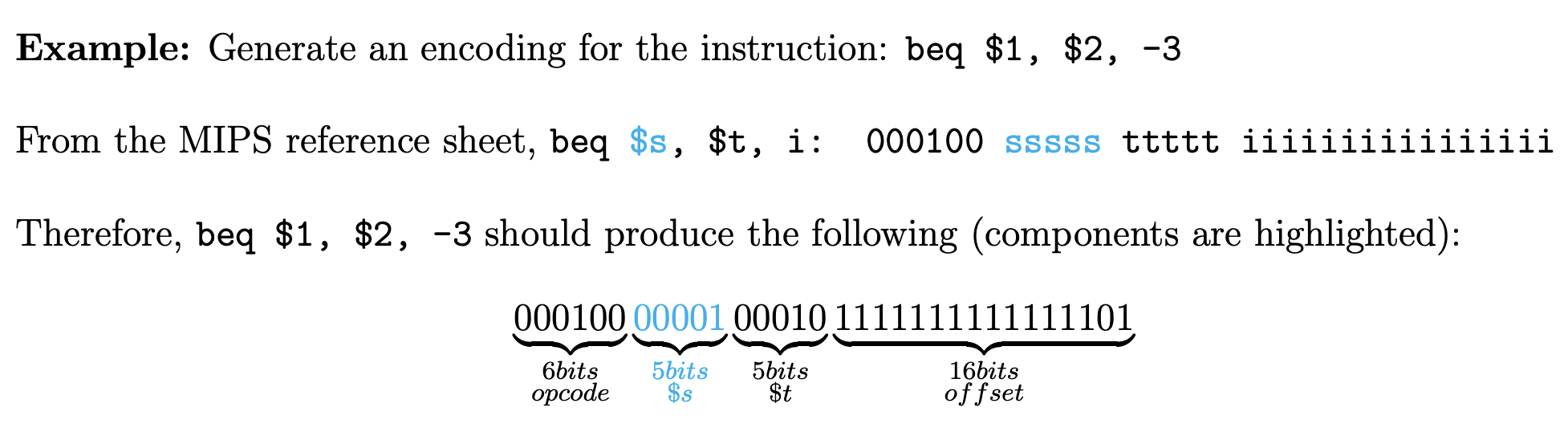
* input to assembler is text file, which MIPS Assembly language program
  + view it as a string of characters
* first break text file down into meaningful tokens such as labels, numbers, .words, MIPS instructions, etc.
  + done using asm.rkt and asm.cc
  + tokenization is often referred to as Scanning stage
  + if tokens aren’t valid, output ERROR to stderr
  + e.g. loop: add $1, $2, $3 can be converted into stream of tokens: LABEL ID REG COMMA REG COMMA REG
* when writing an assembler, dealing with labels can present issues because labels are assembler directives and don’t exist in machine language
  + assembler must convert each use of label into either an address (if used with .word) or offset (if used with beq or bne)
  + below is 2 valid uses of labels:

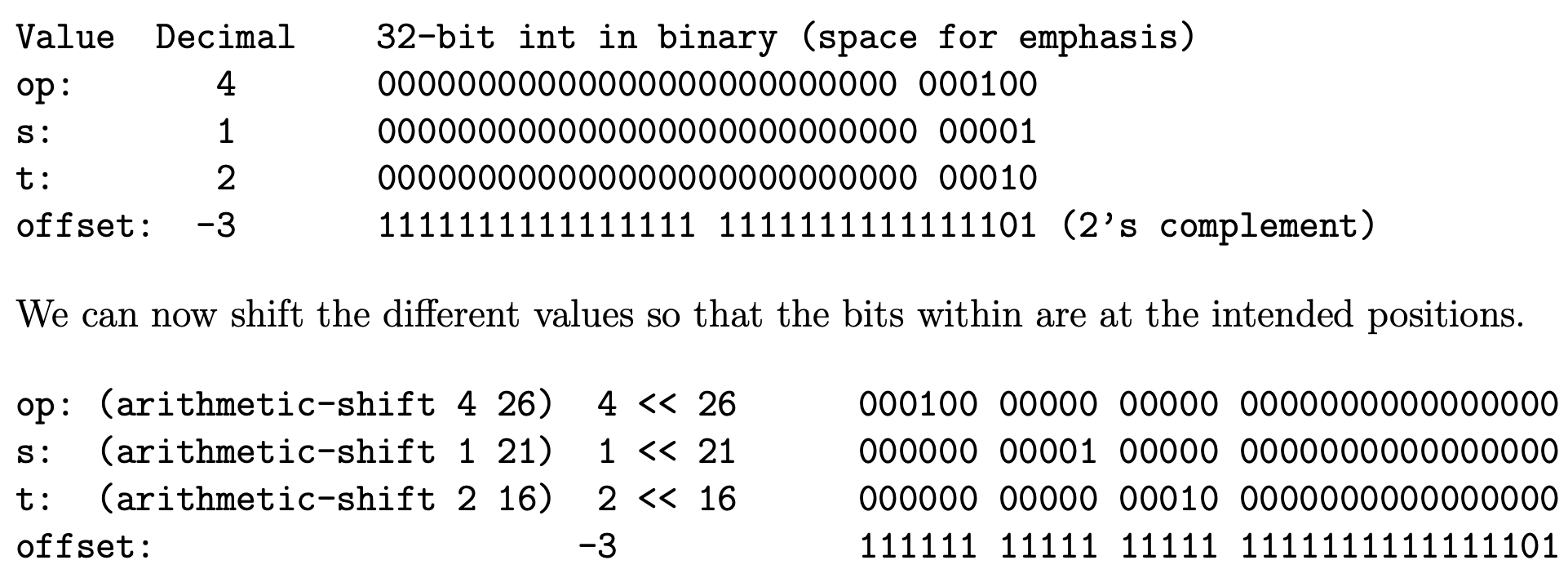


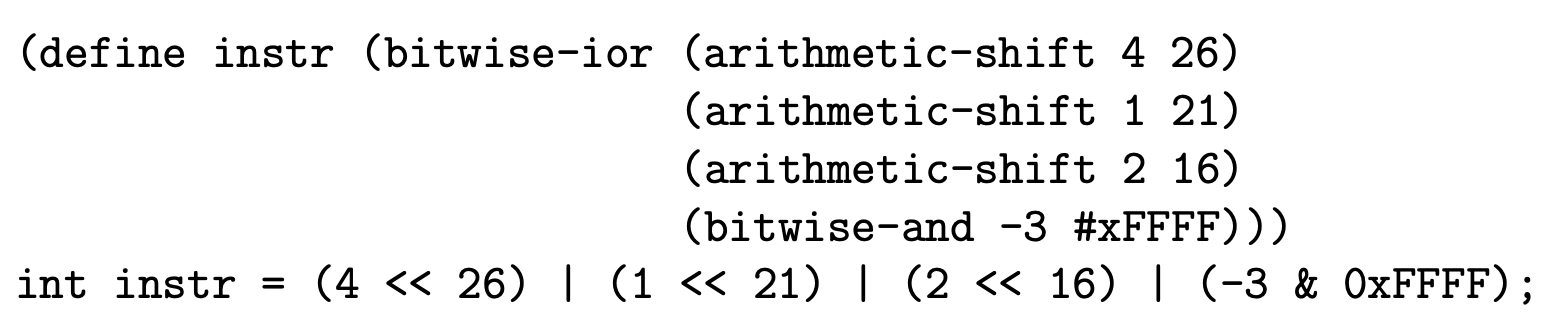
* + - use of label on right is a problem because it’s used before it’s declared
* solution is to implement the assembler in 2 passes
  + pass 1: scanning and parsing
    - group tokens into instructions and record addresses of labels
    - generate symbol table
    - perform duplicate label checks
    - note that multiple labels are possible for the same line
  + pass 2: semantic analysis and synthesis
    - translate each instruction into machine code
    - if instruction refers to a label, look up associated address and process accordingly
    - check labels that are being used are actually defined
* below is example of symbol table that MIPS assembler would create for the program:



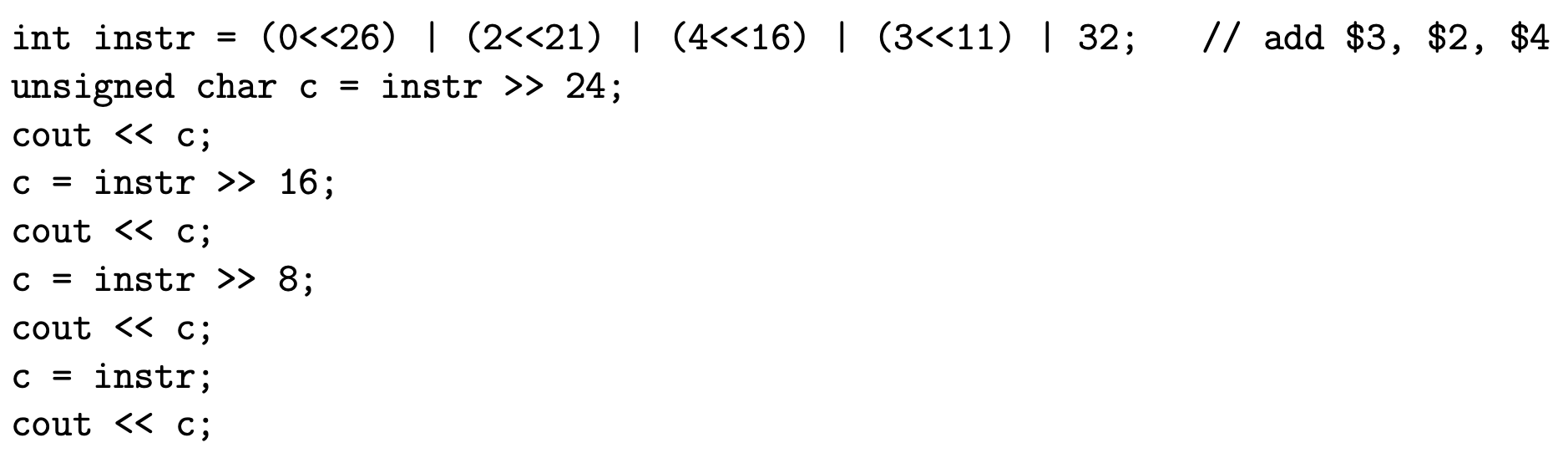
* + when top is used by bne $2, $0, top, assembler must compute offset value
    - offset = (defn − PC) / 4 = (0x14 − 0x20) / 4 = (20 − 32) / 4 = −3
* when writing an assembler:
  + output machine code coming from assembled MIPS code to stdout
  + output symbol table to stderr
* if assembly program is valid, assembler must produce MIPS machine language equivalent on standard output in 2 main steps:
  + first, create encoding for the instructions
  + then, send this output to stdout
* to encode instructions, use bit shifting to put information into correct position, and use bitwise OR to join them
* when dealing with negative values in the offset, use a mask to clear the 16 most significant bits
  + this means applying bitwise AND with 0xFFFF to change the leading 1s to 0s
* below is example of how to encode instructions



* + assuming the assembler has the values of op, s, t, and offset ready:
  + making sure to apply a mask to the –3 in our offset, we can encode the instructions as follows:



* if we wrote cout << instr, then 9 bytes corresponding to the ASCII code for each digit of the instruction would be outputted
  + must use an unsigned char to output the actual bits
* to output machine code instructions:



* + each time, bit shift instr the appropriate amount of places to the right so that the byte we want outputted is in the 8 least significant bits position
  + if concerned about which byte will get copied over when doing the last byte, can mask by doing & 0xFF