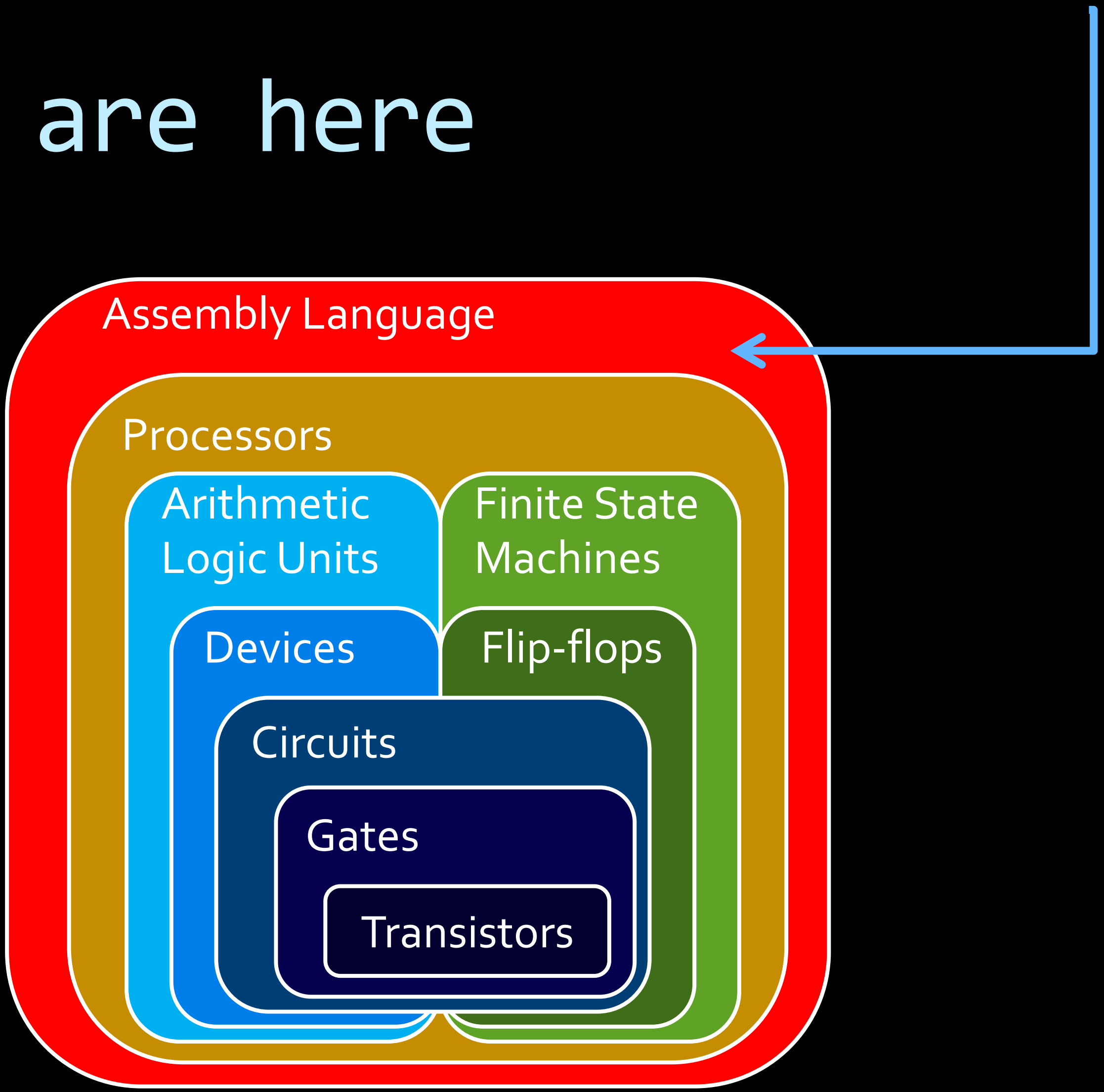
CSC258 Week 9

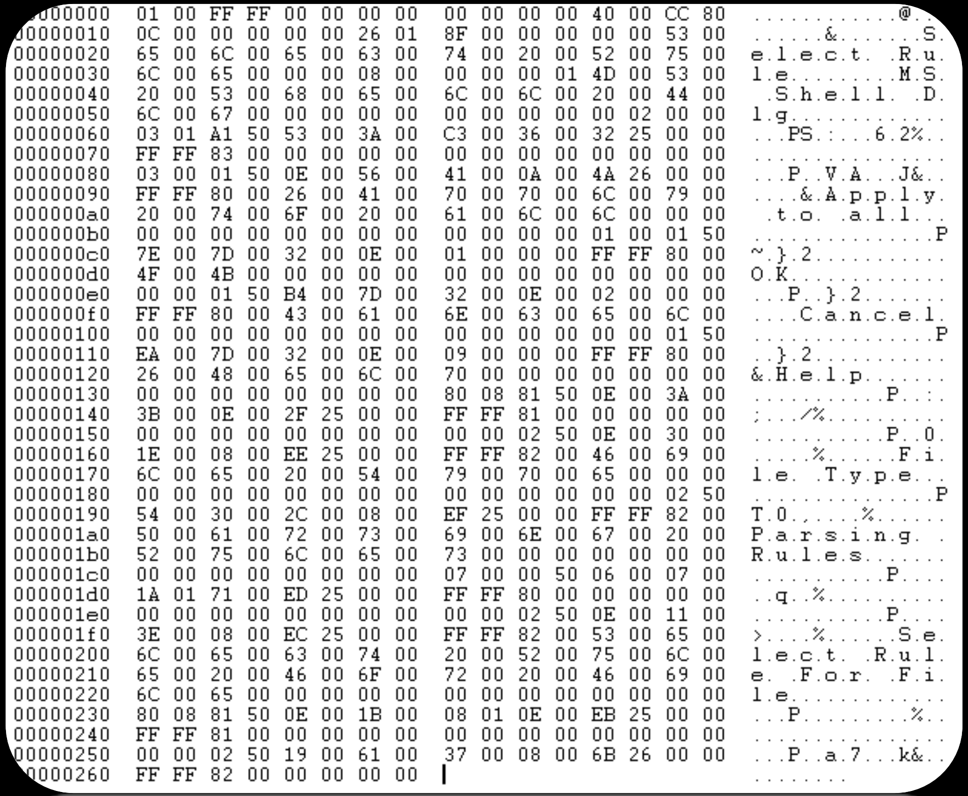
We are here


Assembly Language
Processors
Arithmetic Logic Units
Finite State Machines
Devices
Flip-flops
Circuits
Gates
Transistors

Programming the processor

- Things to learn:
  - Control unit signals to the datapath
  - Machine code instructions
  - Assembly language instructions
  - Programming in assembly language

Machine Code Instructions



```
000000   01 00 FF FF 00 00 00 00  00 00 00 00 40 00 CC 80  ............@...
00000010 0C 00 00 00 00 00 26 01  8F 00 00 00 00 00 53 00  ......&.......S.
00000020 65 00 6C 00 65 00 63 00  74 00 20 00 52 00 75 00  e.l.e.c.t. .R.u.
00000030 6C 00 65 00 00 00 08 00  00 00 00 01 4D 00 53 00  l.e.........M.S.
00000040 20 00 53 00 68 00 65 00  6C 00 6C 00 20 00 44 00   .S.h.e.l.l. .D.
00000050 6C 00 67 00 00 00 00 00  00 00 00 00 00 02 00 00  l.g.............
00000060 03 01 A1 50 53 00 3A 00  C3 00 36 00 32 25 00 00  ...PS.:...6.2%..
00000070 FF FF 83 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000080 03 00 01 50 0E 00 56 00  41 00 0A 00 4A 26 00 00  ...P..V.A...J&..
00000090 FF FF 80 00 26 00 41 00  70 00 70 00 6C 00 79 00  ....&.A.p.p.l.y.
000000a0 20 00 74 00 6F 00 20 00  61 00 6C 00 6C 00 00 00   .t.o. .a.l.l...
000000b0 00 00 00 00 00 00 00 00  00 00 00 00 01 00 01 50  ...............P
000000c0 7E 00 7D 00 32 00 0E 00  01 00 00 00 FF FF 80 00  ~.}.2...........
000000d0 4F 00 4B 00 00 00 00 00  00 00 00 00 00 00 00 00  O.K.............
000000e0 00 00 01 50 B4 00 7D 00  32 00 0E 00 02 00 00 00  ...P..}.2.......
000000f0 FF FF 80 00 43 00 61 00  6E 00 63 00 65 00 6C 00  ....C.a.n.c.e.l.
00000100 00 00 00 00 00 00 00 00  00 00 00 00 00 00 01 50  ...............P
00000110 EA 00 7D 00 32 00 0E 00  09 00 00 00 FF FF 80 00  ..}.2...........
00000120 26 00 48 00 65 00 6C 00  70 00 00 00 00 00 00 00  &.H.e.l.p.......
00000130 00 00 00 00 00 00 00 00  80 08 81 50 0E 00 3A 00  ...........P..:.
00000140 3B 00 0E 00 2F 25 00 00  FF FF 81 00 00 00 00 00  ;.../%..........
00000150 00 00 00 00 00 00 00 00  00 00 02 50 0E 00 30 00  ...........P..0.
00000160 1E 00 08 00 EE 25 00 00  FF FF 82 00 46 00 69 00  .....%......F.i.
00000170 6C 00 65 00 20 00 54 00  79 00 70 00 65 00 00 00  l.e. .T.y.p.e...
00000180 00 00 00 00 00 00 00 00  00 00 00 00 00 00 02 50  ...............P
00000190 54 00 30 00 2C 00 08 00  EF 25 00 00 FF FF 82 00  T.0.,....%......
000001a0 50 00 61 00 72 00 73 00  69 00 6E 00 67 00 20 00  P.a.r.s.i.n.g. .
000001b0 52 00 75 00 6C 00 65 00  73 00 00 00 00 00 00 00  R.u.l.e.s.......
000001c0 00 00 00 00 00 00 00 00  07 00 00 50 06 00 07 00  ...........P....
000001d0 1A 01 71 00 ED 25 00 00  FF FF 80 00 00 00 00 00  ..q..%..........
000001e0 00 00 00 00 00 00 00 00  00 00 02 50 0E 00 11 00  ...........P....
000001f0 3E 00 08 00 EC 25 00 00  FF FF 82 00 53 00 65 00  >....%......S.e.
00000200 6C 00 65 00 63 00 74 00  20 00 52 00 75 00 6C 00  l.e.c.t. .R.u.l.
00000210 65 00 20 00 46 00 6F 00  72 00 20 00 46 00 69 00  e. .F.o.r. .F.i.
00000220 6C 00 65 00 00 00 00 00  00 00 00 00 00 00 00 00  l.e.............
00000230 80 08 81 50 0E 00 1B 00  08 01 0E 00 EB 25 00 00  ...P.........%..
00000240 FF FF 81 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00000250 00 00 02 50 19 00 61 00  37 00 08 00 6B 26 00 00  ...P..a.7...k&..
0000260  FF FF 82 00 00 00 00 00                           ........
```

Intro to Machine Code

- Machine code are the 32-bit binary instructions which the processor can understand (you can now understand, too)
- All programs (C, Java, Python) are eventually translated into machine code (by a compiler or interpreter).
- While executing, the instructions of the program are loaded into the instruction register one by one
- For each instruction loaded, the Control Unit reads the opcode and sets the signals to control the datapath, so that the processor works as instructed.

Assembly language

- Each line of assembly code corresponds to one line of 32-bit long machine code.
- Basically, assembly is a user-friendly way to write machine code.
- Example: `C = A + B`
  - Store `A` in `$t1`, `B` in `$t2`, `C` in `$t3`
  - Assembly language instruction:

```
add $t3, $t1, $t2
```

  - Machine code instruction:

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

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

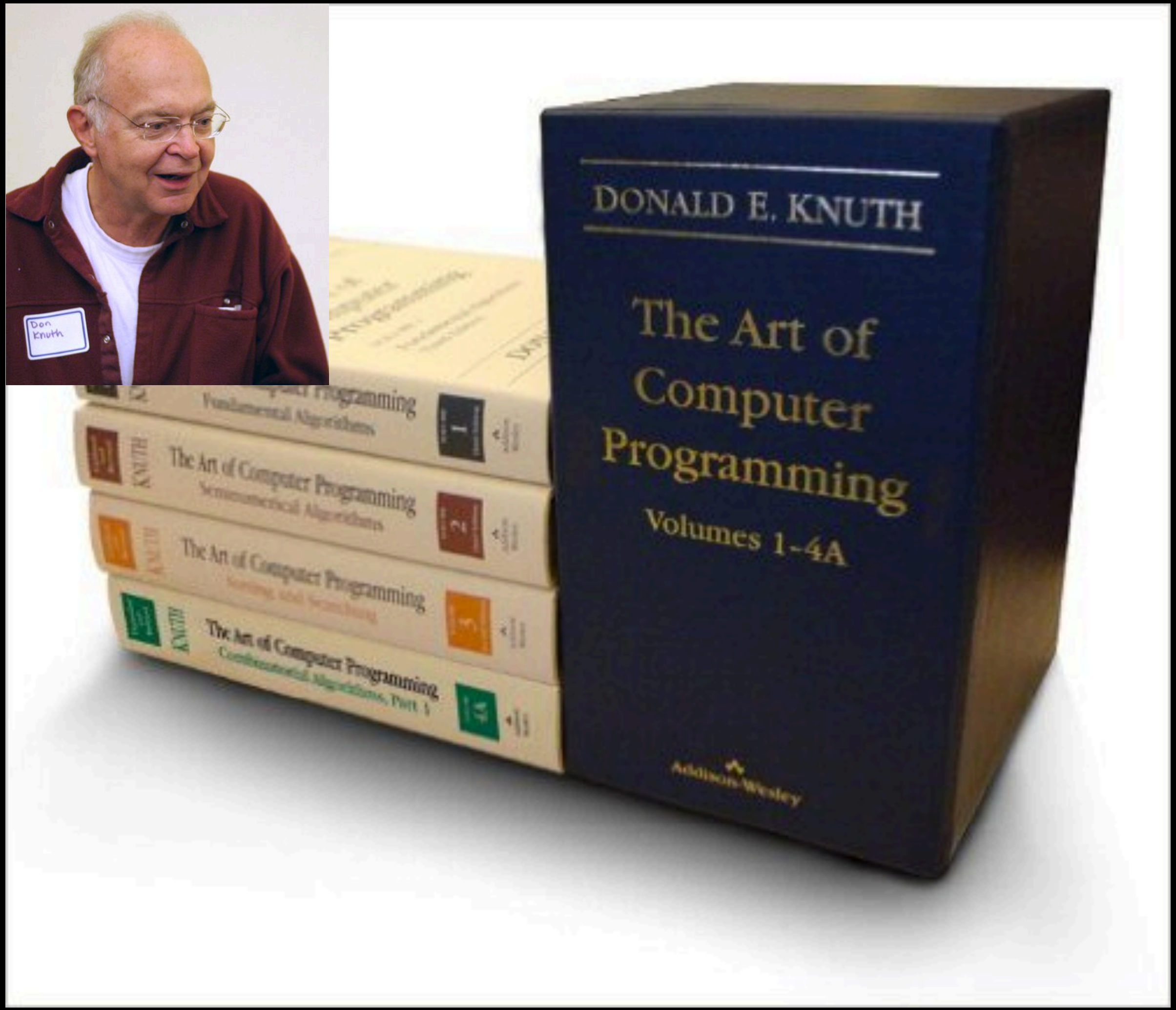
Why learn assembly?

- You'll understand how your program ***really*** works.
- You'll understand your program's performance better by knowing its real "runtime".
- You'll understand how control flows (if / else / for / while) are implemented.
- You'll understand why eliminating if statements makes your code faster.
- You'll understand why pointer is such a natural concept for programming.
- You'll understand the cost of making function calls.
- You'll understand why stack can overflow
- You'll understand there is no "recursion" in the hardware, and how it's actually done.
- You'll understand why memory need to be managed.
- You'll understand why people spend so much time creating operating systems.
- You'll appreciate more the constructs in high-level programming languages.
- And much more…

And, you'll be able to read this book.

Donald Knuth "The Art of Computer Programming"

"All algorithms in this book are written in assembly for clarity."



About register names

- In machine code with have register `0` to register `31`, specified by 5 bits of the instruction.
- In assembly we have names like `$t1`, `$t2`, `$s1`, `$v0`, etc.
- What's the relation between these two?

Machine code + registers

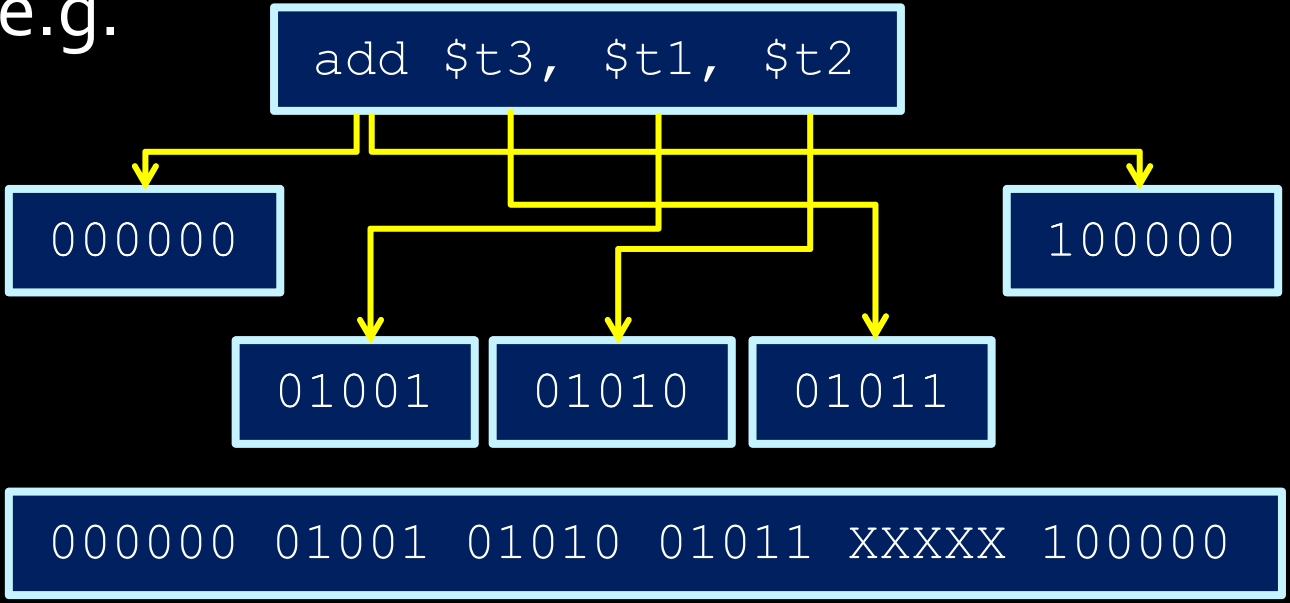
- MIPS is register-to-register.
  - Every operation operates on data in registers.
- MIPS provides 32 registers.
  - Several have special values (conventions):
    - Register `0` ($zero): value `0` -- always.
    - Register `1` ($at): reserved for the assembler.
    - Registers `2-3` ($v0, $v1): return values
    - Registers `4-7` ($a0-$a3): function arguments
    - Registers `8-15`, `24-25` ($t0-$t9): temporaries
    - Registers `16-23` ($s0-$s7): saved temporaries
    - Registers `28-31` ($gp, $sp, $fp, $ra): memory and function support
    - Registers `26-27`: reserved for OS kernel
  - Also three special registers (PC, HI, LO) that are not directly accessible.
    - HI and LO are used in multiplication and division, and have special instructions for accessing them.

$v0, $t2, $a3, etc are the registers' nicknames in assembly

Technically you can use any register for anything, but this is the convention

Translate assembly to machine code

- When writing machine code instructions (or interpreting them), we need to know which register values to encode (or decode).
- e.g.



Machine code details

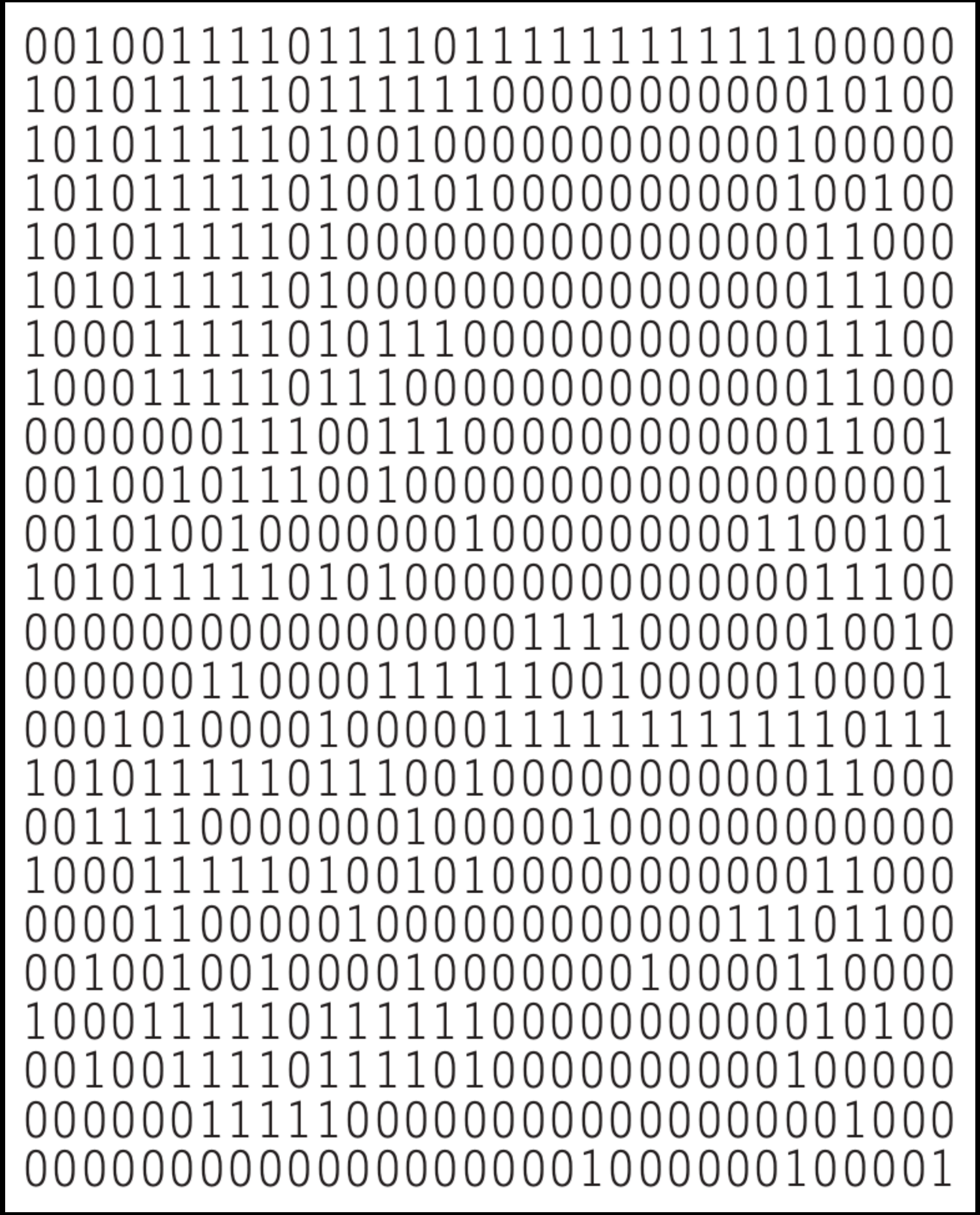
- Things to note about machine code:
  - R-type instructions have an opcode of `000000`, with a 6-bit function listed at the end.
  - Although we specify "don't care" bits as `X` values, the assembly language interpreter always assigns them to some value (like `0`)
  - In exams, we want you to write X instead of `0`, to show that you know we don't care those bits

Try this at home

Below is the content of an executable file "mystery.exe", what does this program do?

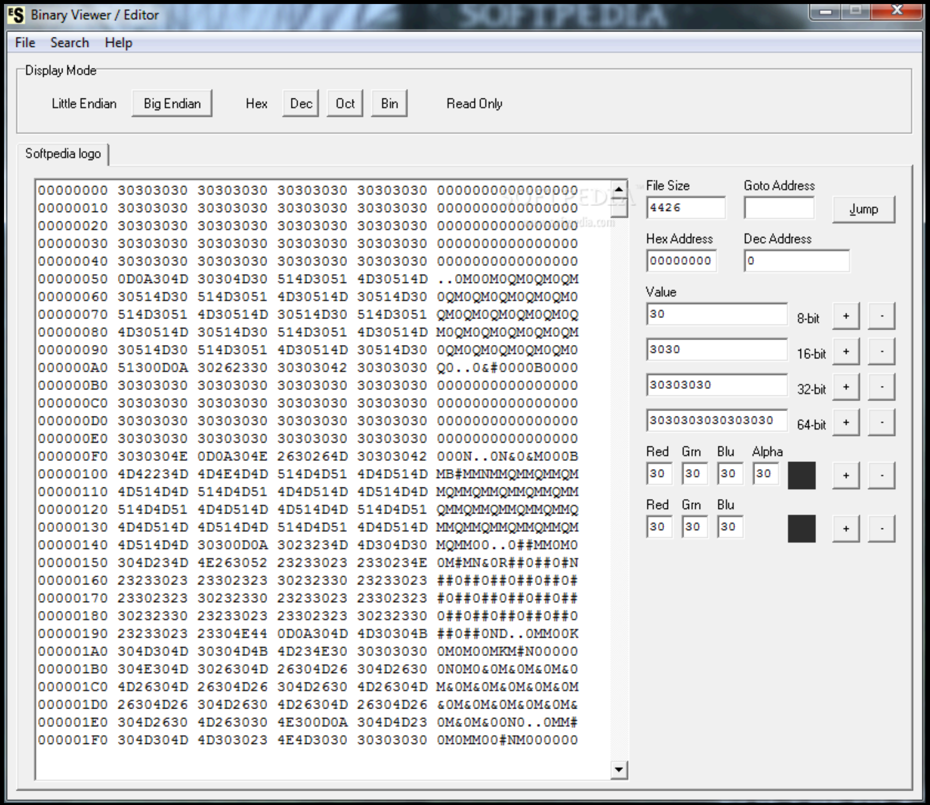
```
1000 1110 0000 1000 0101 1010 1111 0001
1000 1110 0010 1001 1101 0010 0011 0010
0000 0001 0000 1001 0101 0000 0010 0000
1000 1110 0100 1011 1111 0011 0011 0111
0000 0000 0000 1100 0011 0001 0000 0000
0000 0010 0110 1010 1010 0000 0010 0010
1010 1101 1101 0100 0000 1111 0101 1010
```

Then try this one



```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000111100000010010
00000011000011111100100000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

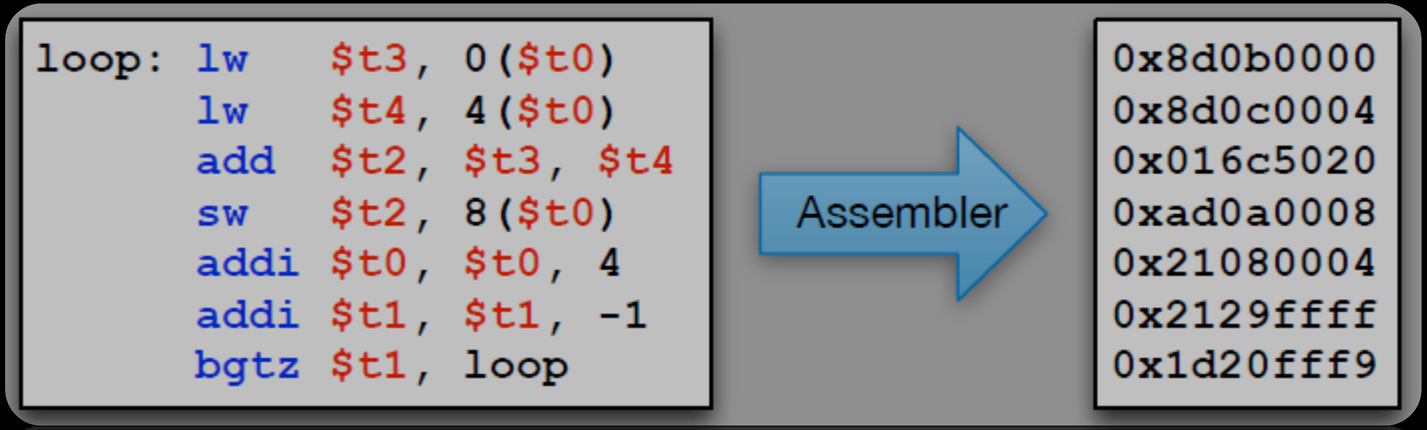
Now you can totally program an executable like this (don’t even need a compiler).



```
00000000 30303030 30303030 30303030 30303030 0000000000000000
00000010 30303030 30303030 30303030 30303030 0000000000000000
00000020 30303030 30303030 30303030 30303030 0000000000000000
00000030 30303030 30303030 30303030 30303030 0000000000000000
00000040 30303030 30303030 30303030 30303030 0000000000000000
00000050 0D0A304D 30304D30 514D3051 4D30514D ..0M00M0QM0QM0QM
00000060 30514D30 514D3051 4D30514D 30514D30 0QM0QM0QM0QM0QM0
00000070 514D3051 4D30514D 30514D30 514D3051 QM0QM0QM0QM0QM0Q
00000080 4D30514D 30514D30 514D3051 4D30514D M0QM0QM0QM0QM0QM
00000090 30514D30 514D3051 4D30514D 30514D30 0QM0QM0QM0QM0QM0
000000A0 51300D0A 30262330 30303042 30303030 Q0..0&#0000B0000
000000B0 30303030 30303030 30303030 30303030 0000000000000000
000000C0 30303030 30303030 30303030 30303030 0000000000000000
000000D0 30303030 30303030 30303030 30303030 0000000000000000
000000E0 30303030 30303030 30303030 30303030 0000000000000000
000000F0 3030304E 0D0A304E 2630264D 30303042 000N..0N&0&M000B
00000100 4D42234D 4D4E4D4D 514D4D51 4D4D514D MB#MMNMMQMMQMMQM
00000110 4D514D4D 514D4D51 4D4D514D 4D514D4D MQMMQMMQMMQMMQMM
00000120 514D4D51 4D4D514D 4D514D4D 514D4D51 QMMQMMQMMQMMQMMQ
00000130 4D4D514D 4D514D4D 514D4D51 4D4D514D MMQMMQMMQMMQMMQM
00000140 4D514D4D 30300D0A 3023234D 4D304D30 MQMM00..0##MM0M0
00000150 304D234D 4E263052 23233023 2330234E 0M#MN&0R##0##0#N
00000160 23233023 23302323 30232330 23233023 ##0##0##0##0##0#
00000170 23302323 30232330 23233023 23302323 #0##0##0##0##0##
00000180 30232330 23233023 23302323 30232330 0##0##0##0##0##0
00000190 23233023 23304E44 0D0A304D 4D30304B ##0##0ND..0MM00K
000001A0 304D304D 30304D4B 4D234E30 30303030 0M0M00MKM#N00000
000001B0 304E304D 3026304D 26304D26 304D2630 0N0M0&0M&0M&0M&0
000001C0 4D26304D 26304D26 304D2630 4D26304D M&0M&0M&0M&0M&0M
000001D0 26304D26 304D2630 4D26304D 26304D26 &0M&0M&0M&0M&0M&
000001E0 304D2630 4D263030 4E300D0A 304D4D23 0M&0M&00N0..0MM#
000001F0 304D304D 4D303023 4E4D3030 30303030 0M0MM00#NM000000
```

File Size: 4426
Goto Address:
Hex Address: 00000000
Dec Address: 0
Value:
8-bit: 30
16-bit: 3030
32-bit: 30303030
64-bit: 3030303030303030
Red: 30 Grn: 30 Blu: 30 Alpha: 30
Red: 30 Grn: 30 Blu: 30

Assembly Language Instructions

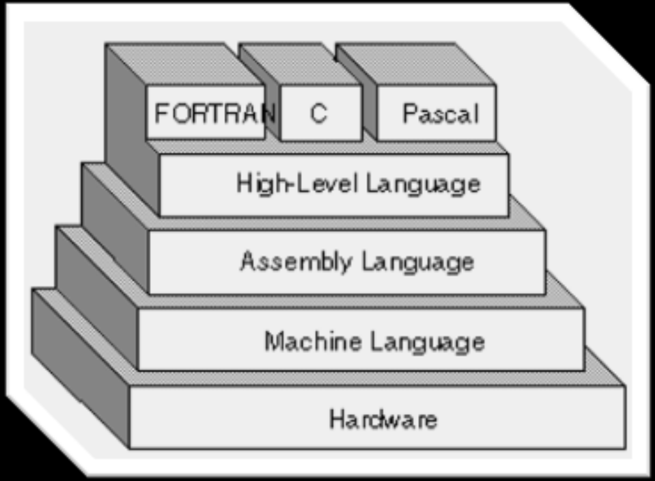


```
loop: lw   $t3, 0($t0)
      lw   $t4, 4($t0)
      add  $t2, $t3, $t4
      sw   $t2, 8($t0)
      addi $t0, $t0, 4
      addi $t1, $t1, -1
      bgtz $t1, loop
```

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

Assembly language

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



Trivia

The thing that converts assembly code to executable is NOT called a compiler.

It's called an assembler, because there is no fancy complication needed, it just assembles the lines!

You should be able to write one easily (e.g., for the processor created in Lab 7)

A little about MIPS

- MIPS
  - Short for Microprocessor without Interlocked Pipeline Stages
    - A type of RISC (Reduced Instruction Set Computer) architecture.
  - Provides a set of simple and fast instructions
    - Compiler translates instructions into 32-bit instructions for instruction memory.
    - Complex instructions are built out of simple ones by the compiler and assembler.

The layout of assembly code

Code sectioning syntax: example

```
.data
A:         .space    400       # array of 100 integers
B:         .space    400       # array of 100 integers

.text
main:      add $t0, $zero, $zero     # load “0” into $t0
           addi $t1, $zero, 400      # load “400" into $t1
           addi $t9, $zero, B        # store address of B
           addi $t8, $zero, A        # store address of A

loop:      add $t4, $t8, $t0  # $t4 = addr(A) + i
           add $t3, $t9, $t0  # $t3 = addr(B) + i
           lw $s4, 0($t3)     # $s4 = B[i]
           addi $t6, $s4, 1   # $t6 = B[i] + 1
           sw $t6, 0($t4)     # A[i] = $t6
           addi $t0, $t0, 4   # $t0 = $t0++
           bne $t0, $t1, loop # branch back if $t0<400
end:
```

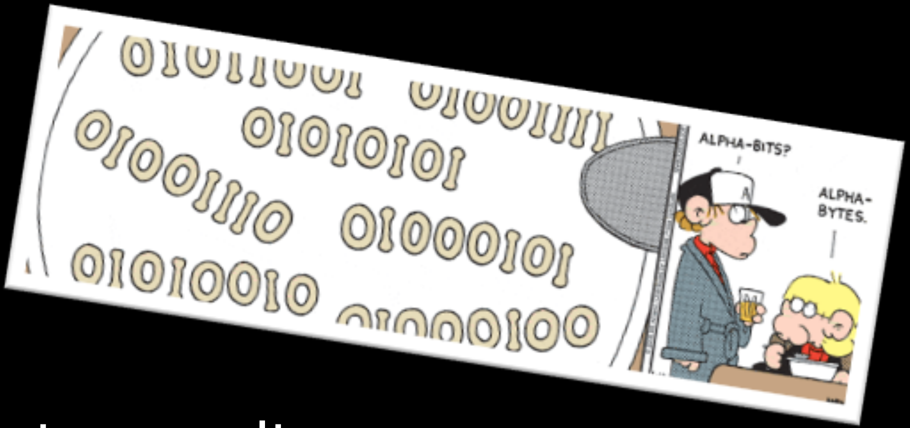
Code sectioning syntax

- `.data`
  - Indicates the start of the data declarations.
- `.text`
  - Indicates the start of the program instructions.
- `main:`
  - The initial line to run when executing the program.

- You can create other labels as needed.

MIPS Instructions

- Things to note about MIPS instructions:
  - Instruction are written as: `<instr> <parameters>`
  - Each instruction is written on its own line
  - All instructions are 32 bits (4 bytes) long
  - Instruction addresses are measured in **bytes**, starting from the instruction at address 0.
- The following tables show the most common MIPS instructions, the syntax for their parameters, and what operation they perform.



Arithmetic instructions

Instruction	Opcode/Function	Syntax	Operation
add	100000	$d, $s, $t	$d = $s + $t
addu	100001	$d, $s, $t	$d = $s + $t
addi	001000	$t, $s, i	$t = $s + SE(i)
addiu	001001	$t, $s, i	$t = $s + SE(i)
div	011010	$s, $t	lo = $s / $t; hi = $s % $t
divu	011011	$s, $t	lo = $s / $t; hi = $s % $t
mult	011000	$s, $t	hi:lo = $s * $t
multu	011001	$s, $t	hi:lo = $s * $t
sub	100010	$d, $s, $t	$d = $s - $t
subu	100011	$d, $s, $t	$d = $s - $t

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

ALU instructions

- Note that for ALU instruction, most are R-type instructions.
  - The six-digit codes in the tables are therefore the function codes (opcodes are `000000`).
  - Exceptions are the I-type instructions (`addi`, `andi`, `ori`, etc.)
- Not all R-type instructions have an I-type equivalent.
  - RISC architectures dictate that an operation doesn't need an instruction if it can be performed through multiple existing operations.
  - Example: `divi $t0, 42` `can be done by`
  - `addi $t1, $zero, 42`
  - `div $t0 $t1`

Logical instructions

Instruction	Opcode/Function	Syntax	Operation
`and`	100100	\$d, \$s, \$t	\$d = \$s & \$t
`andi`	001100	\$t, \$s, i	\$t = \$s & ZE(i)
`nor`	100111	\$d, \$s, \$t	\$d = ~(\$s | \$t)
`or`	100101	\$d, \$s, \$t	\$d = \$s | \$t
`ori`	001101	\$t, \$s, i	\$t = \$s | ZE(i)
`xor`	100110	\$d, \$s, \$t	\$d = \$s ^ \$t
`xori`	001110	\$t, \$s, i	\$t = \$s ^ ZE(i)

Note: ZE = zero extend (pad upper bits with 0 value).

Shift instructions

Instruction	Opcode/Function	Syntax	Operation
`sll`	000000	\$d, \$t, a	\$d = \$t << a
`sllv`	000100	\$d, \$t, \$s	\$d = \$t << \$s
`sra`	000011	\$d, \$t, a	\$d = \$t >> a
`srav`	000111	\$d, \$t, \$s	\$d = \$t >> \$s
`srl`	000010	\$d, \$t, a	\$d = \$t >>> a
`srlv`	000110	\$d, \$t, \$s	\$d = \$t >>> \$s

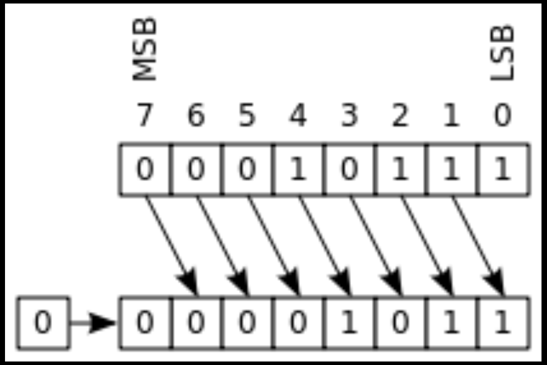
Note: `srl` = "shift right logical", and `sra` = "shift right arithmetic".
The "`v`" denotes a variable number of bits, specified by `$s`.

Logic shift vs Arithmetic shift

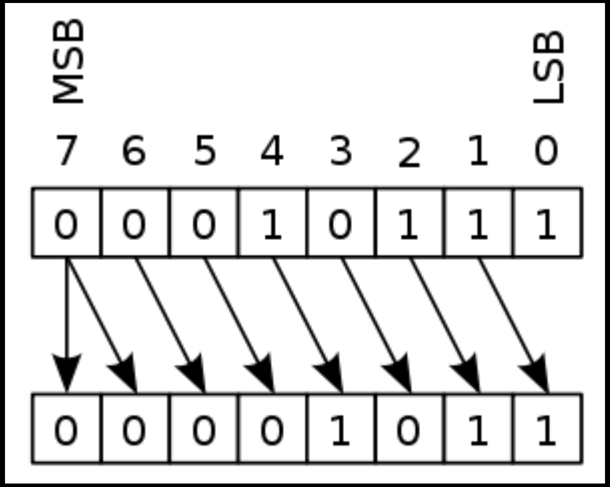
Left shift: same, fill empty spot (lower bits) with zeros
(that's why we have `sll` but no `sla`)

Right shift: different

- Logic shift fills empty spot(higher bits) with zeros
- Arithmetic shift fills empty spot (higher bits) with the MSB of the original number.



Logic



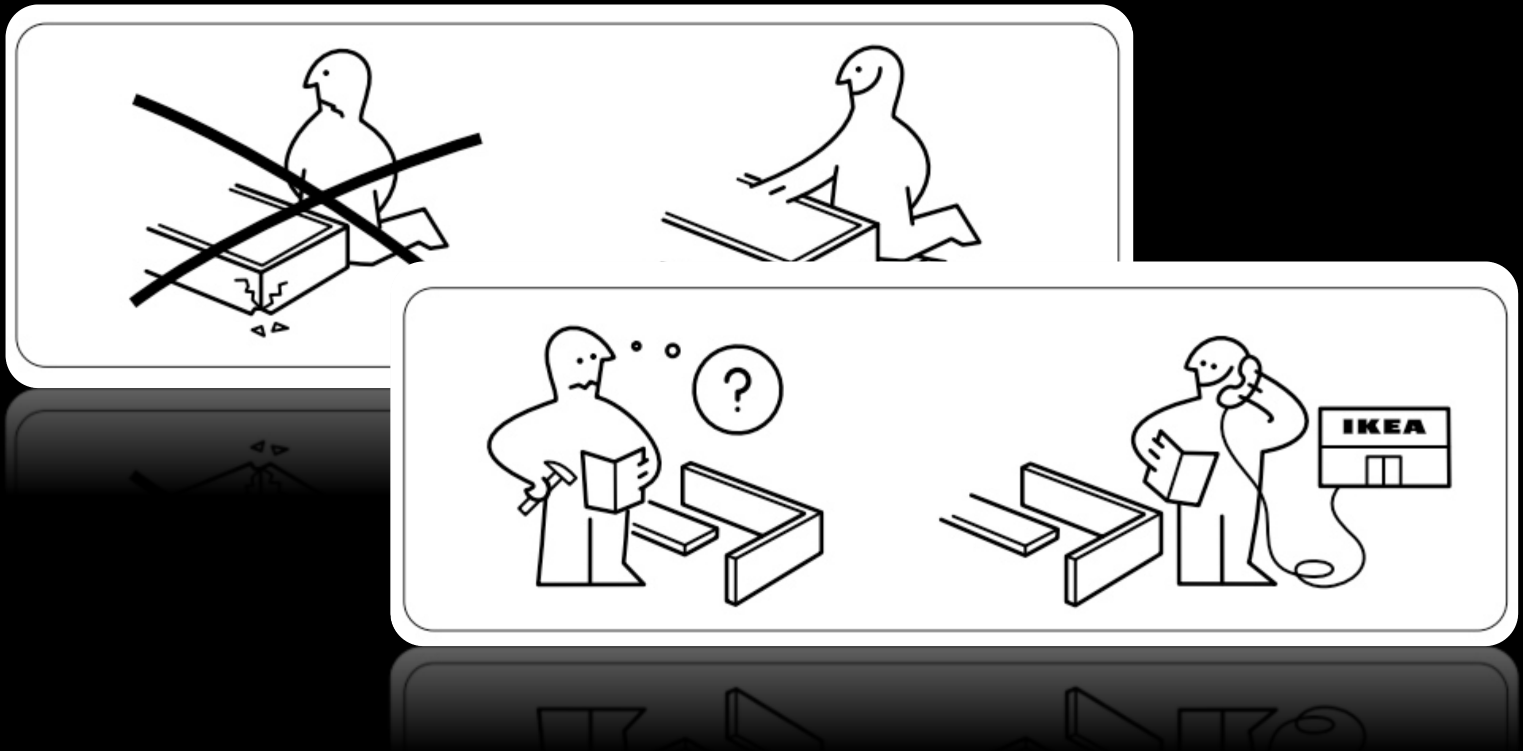
Arithmetic

Data movement instructions

Instruction	Opcode/Function	Syntax	Operation
`mfhi`	010000	$d	$d = hi
`mflo`	010010	$d	$d = lo
`mthi`	010001	$s	hi = $s
`mtlo`	010011	$s	lo = $s

- These are instructions for operating on the HI and LO registers described earlier.

Time for more instructions!



Flow control:
Branch and loop

Control flow in assembly

- Not all programs follow a linear set of instructions.
  - Some operations require the code to branch to one section of code or another (if/else).
  - Some require the code to jump back and repeat a section of code again (for/while).
- For this, we have labels on the left-hand side that indicate the points that the program flow might need to jump to.
  - References to these points in the assembly code are resolved at compile time to offset values for the program counter.

Code sectioning syntax: example

```
.data
A:         .space   400       # array of 100 integers
B:         .space   400       # array of 100 integers

.text
main:      add $t0, $zero, $zero      # load “0” into $t0
           addi $t1, $zero, 400       # load “400" into $t1
           addi $t9, $zero, B         # store address of B
           addi $t8, $zero, A         # store address of A

loop:      add $t4, $t8, $t0   # $t4 = addr(A) + i
           add $t3, $t9, $t0   # $t3 = addr(B) + i
           lw $s4, 0($t3)      # $s4 = B[i]
           addi $t6, $s4, 1    # $t6 = B[i] + 1
           sw $t6, 0($t4)      # A[i] = $t6
           addi $t0, $t0, 4    # $t0 = $t0++
           bne $t0, $t1, loop # branch back if $t0<400
end:
```

Branch instructions

Instruction	Opcode/Function	Syntax	Operation
`beq`	000100	$s, $t, label	if ($s == $t) pc += i << 2
`bgtz`	000111	$s, label	if ($s > 0) pc += i << 2
`blez`	000110	$s, label	if ($s <= 0) pc += i << 2
`bne`	000101	$s, $t, label	if ($s != $t) pc += i << 2

- Branch operations are key when implementing if statements and while loops.
- The labels are memory locations, assigned to each label at compile time.
    - Note: `i` is calculated as `(label - (current PC + 4)) >> 2`

Comparison instructions

Instruction	Opcode/Function	Syntax	Operation
`slt`	101010	\$d, \$s, \$t	\$d = (\$s < \$t)
`sltu`	101001	\$d, \$s, \$t	\$d = (\$s < \$t)
`slti`	001010	\$t, \$s, i	\$t = (\$s < SE(i))
`sltiu`	001001	\$t, \$s, i	\$t = (\$s < SE(i))

Note: Comparison operation stores a one in the destination register if the less-than comparison is true, and stores a zero in that location otherwise.

Note: Real vs Pseudo instructions

What we list in the slides are all real instructions, i.e., each one has an **opcode** corresponding to it.

There are some pseudo-instructions, which don’t have their own opcode, but is implemented using real instructions; they are provided for coding convenience.

For example:

- `bge $t0,$t1,Label` is actually
- `slt $t2,$t0,$t1;  beq $t2,$zero,Label`

Jump instructions

Instruction	Opcode/Function	Syntax	Operation
`j`	000010	label	pc += i << 2
`jal`	000011	label	\$31 = pc; pc += i << 2
`jalr`	001001	\$s	\$31 = pc; pc = \$s
`jr`	001000	\$s	pc = \$s

- `jal` = "jump and link".
  - Register \$31 (aka \$ra) stores the address that's used when returning from a subroutine.
- Note: `jr` and `jalr` are not j-type instructions.

If/Else statements in MIPS

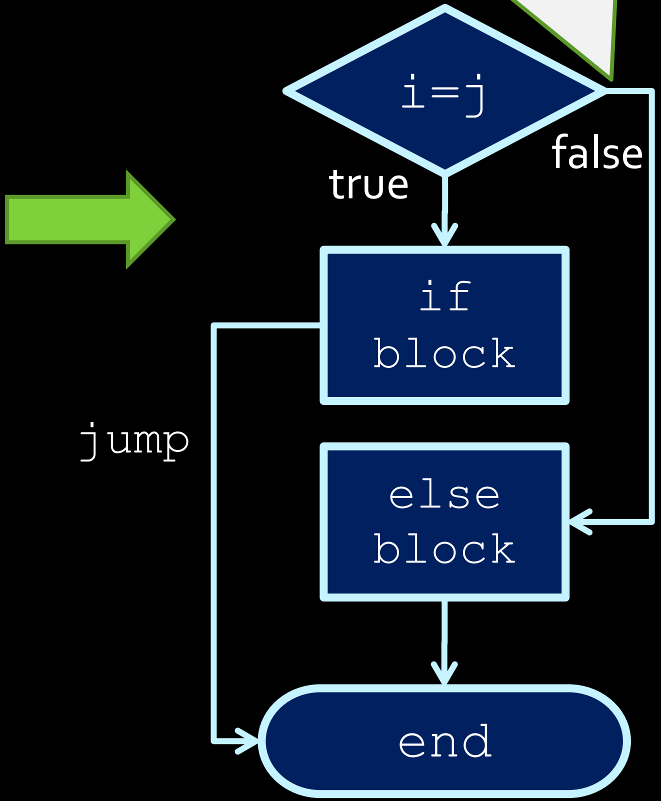
```
if ( i == j )
  i++;
else
  j--;
j += i;
```

- Strategy for if/else statements:
  - Test condition, and jump to `if` logic block whenever condition is true.
  - Otherwise, perform `else` logic block, and jump to first line after `if` logic block.
- A flowchart can be helpful here

If statement

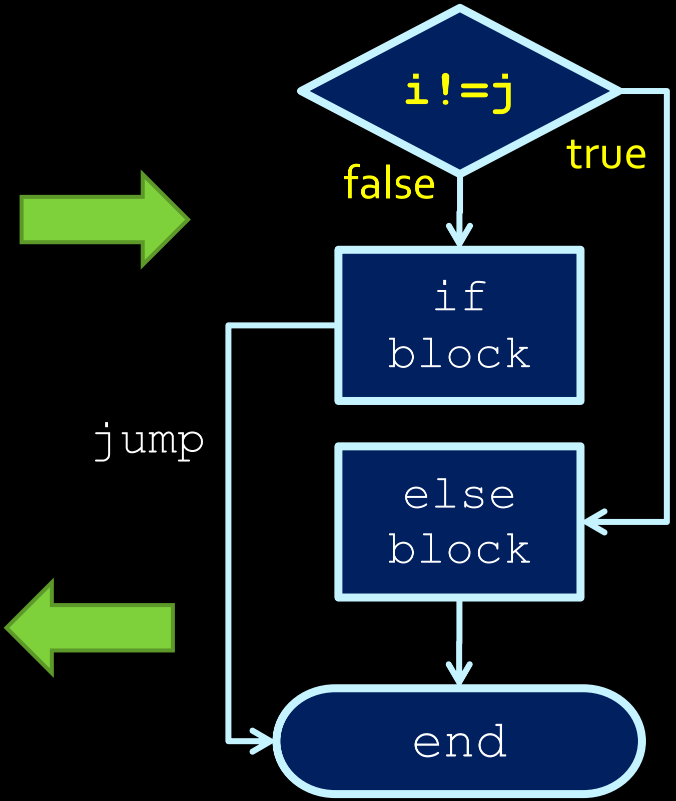
Only problem: branch instructions jump on TRUE instead of FALSE, so negate the checked condition to i != j

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



If statement flowcharts

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



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

Translated if/else statements

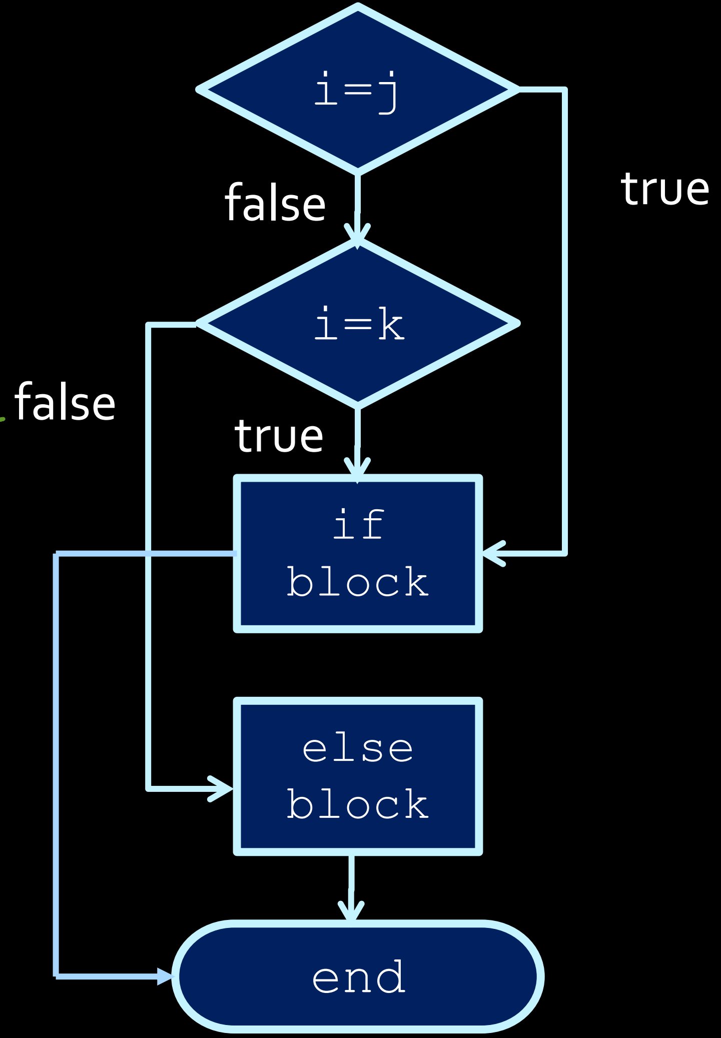
```
#   $t1 = i, $t2 = j
main:     bne  $t1, $t2, ELSE    # branch if ! ( i == j )
          addi $t1, $t1, 1       # i++
          j END                  # jump over ELSE
ELSE:     addi $t2, $t2, -1      # j--
END:      add $t2, $t2, $t1      # j += i
```

- If we change BNE to BEQ, then we also need to swap the IF and ELSE blocks

```
#   $t1 = i, $t2 = j
main:     beq  $t1, $t2, IF    # branch if ( i == j )
          addi $t2, $t2, -1    # j--
          j END                # jump over IF
IF:       addi $t1, $t1, 1     # i++
END:      add $t2, $t2, $t1    # j += i
```

Multiple if conditions

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

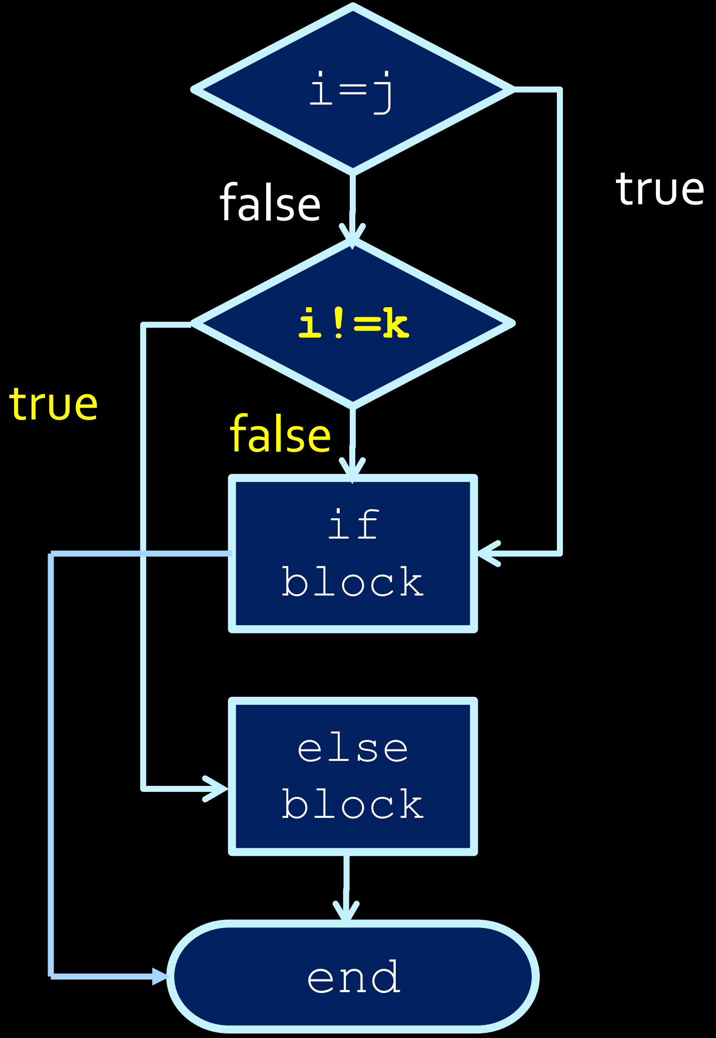


Branch on FALSE!

Multiple if conditions

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

```
#  $t1 = i, $t2 = j, $t3 = k
main:  beq $t1, $t2, IF
       bne $t1, $t3, ELSE
IF:    addi $t1, $t1, 1
       j END
ELSE:  addi $t2, $t2, -1
END:   add $t2, $t1, $t3
```



Loops

Loops in MIPS (while loop)

- Example of a simple loop, in assembly:

```
#  $t0 = i, $t1 = n
main:    add $t0, $zero, $zero   # i = 0
         addi $t1, $zero, 100    # n = 100
START:   beq $t0, $t1, END       # if i == n, END
         addi $t0, $t0, 1        # i++
         j START
END:
```

- ...which is the same as saying (in C):

```
while (i != 100) {
   i++;
}
```

For loop

```
for ( <init> ; <cond> ; <update> ) {
    <for body>
}
```

- For loops (such as above) are usually implemented with the following structure:

```
main:     <init>
START:    if (!<cond>) branch to END
           <for-body>
UPDATE:    <update>
           jump to START
END:
```

Exercise:

```
j = 0
for ( _____ ; _____ ; _____ )
{
     j = j + i;
}
```

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

Answer

```
j = 0
for ( i=0 ; i!=100 ; i++ )
{
     j = j + i;
}
```

- This translates to:

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

- `while` loops are the same, without the initialization and update sections.

Another exercise

- Fibonacci sequence:
  - How would you convert this into assembly?

```
int fib(void) {
   int n = 10;
   int f1 = 1, f2 = -1;

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

Assembly code example

- Fibonacci sequence in assembly code:

```
int fib(void) {
   int n = 10;
   int f1 = 1, f2 = -1;

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

```
# fib.asm
# register usage: $t3=n, $t4=f1, $t5=f2
# RES refers to memory address of result
FIB:  addi $t3, $zero, 10     # initialize n=10
      addi $t4, $zero, 1      # initialize f1=1
      addi $t5, $zero, -1     # initialize f2=-1
LOOP: beq $t3, $zero, END     # done loop if n==0
      add $t4, $t4, $t5       # f1 = f1 + f2
      sub $t5, $t4, $t5       # f2 = f1 - f2
      addi $t3, $t3, -1       # n = n - 1
      j LOOP                  # repeat until done
END:  sb $t4, RES             # store result
```

Making an assembly program

- Assembly language programs typically have structure similar to simple Python or C programs:
  - They set aside registers to store data.
  - They have sections of instructions that manipulate this data.
- It is always good to decide at the beginning which registers will be used for what purpose!
  - More on this later ☺