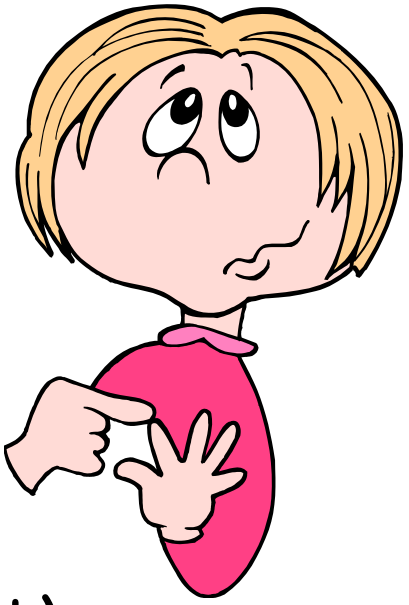
# Adventures in Assembly Land

- What is an Assembler
- ASM Directives
- ASM Syntax
- Intro to MARS
- Simple examples

# A Simple Programming Task

- Add the numbers 0 to 4 ...
- 10 = 0 + 1 + 2 + 3 + 4
- In "C":



```
int i, sum;

main() {
     sum = 0;
     for (i=0; i<5; i++)
          sum = sum + i;
}
```

- Now let's code it in ASSEMBLY

# What IS an Assembler?

- A program for writing programs
- Machine Language:
  - 1's and 0's loaded into memory.
    (Did anybody ever really do that?)
- Assembly Language:

Front panel of a classic PDP8e. The toggle switches were used to enter machine language.

```
    .globl main
main:
    subu $sp, $sp, 24
    sw   $ra, 16($sp)
    li   $a0, 18
    li   $a1, 12
    li   $a2, 6
    jal  tak
    move $a0, $v0
```

Symbolic SOURCE text file

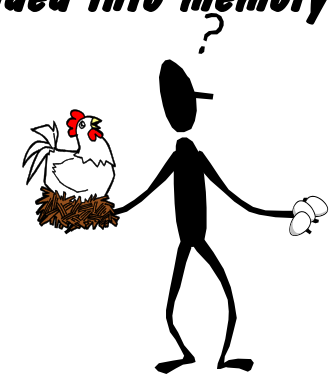
ASM

ASSEMBLER Translator program

```
01101101
11000110
00101111
10110001
.....
```

Binary Machine Language

***STREAM of bits to be loaded into memory***



**Assembler:**

1. A Symbolic LANGUAGE for representing strings of bits
2. A PROGRAM for translating Assembly Source to binary.

# Assembly Source Language

An Assembly SOURCE FILE contains, in symbolic text, values of successive bytes to be loaded into memory... e.g.

```
.data 0x10000000
.byte 1, 2, 3, 4
.byte 5, 6, 7, 8
.word 1, 2, 3, 4
.asciiz "Comp 411"
.align 2
.word 0xfeedbeef
```

- Specifies address where following values are loaded
- First four byte values
- Second four byte values
- Four WORD values (each is 4 bytes)
- A string of 9 ASCII bytes (8 + NULL)
- Align to next multiple of 4 ($2^2$)
- Hex encoded WORD Value

Resulting memory dump:

```
[0x10000000]    0x04030201  0x08070605  0x00000001  0x00000002
[0x10000010]    0x00000003  0x00000004  0x706d6f43  0x31313420
[0x10000020]    0x00000000  0xfeedbeef  0x00000000  0x00000000
```

Notice the byte ordering. This MIPS is "little-endian" (The least significant byte of a word or half-word has the lowest address)

# Assembler Syntax

- Assembler DIRECTIVES (Keywords prefixed with a '.')
  - Control the placement and interpretation of bytes in memory

| | |
|---|---|
| `.data <addr>` | Subsequent items are considered data |
| `.text <addr>` | Subsequent items are considered instructions |
| `.align N` | Skip to next address multiple of $2^N$ |

  - Allocate Storage

| | |
|---|---|
| `.byte` $b_1, b_2, \ldots, b_n$ | Store a sequence of bytes (8-bits) |
| `.half` $h_1, h_2, \ldots, h_n$ | Store a sequence of half-words (16-bits) |
| `.word` $w_1, w_2, \ldots, w_n$ | Store a sequence of words (32-bits) |
| `.ascii "string"` | Stores a sequence of ASCII encoded bytes |
| `.asciiz "string"` | Stores a zero-terminated string |
| `.space n` | Allocates n successive bytes |

  - Define scope

| | |
|---|---|
| `.globl sym` | Declares symbol to be visible to other files |
| `.extern sym size` | Sets size of symbol defined in another file (Also makes it DIRECTly addressable) |

# More Assembler Syntax

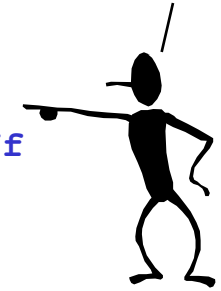
- Assembler COMMENTS

  All text following a '#' (sharp) to the end of the line is ignored

- Assembler LABELS
  - Labels are symbols that represent memory addresses. Labels take on the values of the address where they are declared. Labels declarations appear at the beginning of a line, and are terminated by a colon. Labels can be established for data items as well as instructions… e. g.

```
        .data
item:   .word 1                 # a data word
        .text
start:  .word 0x00821820        # add   $3, $4, $2
        .word 0x00031a00        # sll   $3, $3, 8
        .word 0x306300ff        # andi  $3, $3, 0xff
```

While having an assembler helps, coding like this is still painful. (Don't actually do this!)



# Our Example: Variable Allocation

- Two integer variables (by default 32 bits in MIPS)

```
.data
.globl sum
.globl i
sum:    .space 4
i:      .space 4
```

- ".data" assembler directive places the following words into the data segment
- ".globl" directives make the "sum" and "I" variables visible to all other assembly modules
- ".space" directives allocate 4 bytes for each variable

# Even More Assembler Syntax

- Assembler PREDEFINED SYMBOLS
  - Register names and aliases

    ```
    $0-$31, $zero, $v0-$v1, $a0-$a3, $t0-$t9, $s0-$s7,
    $at, $k0-$k1, $gp, $sp, $fp, $ra
    ```

- Assembler MNEMONICS
  - Symbolic representations of individual instructions

    ```
    add, addu, addi, addiu, sub, subu,
    and, andi, or, ori, xor, xori, nor, lui,
    sll, sllv, sra, srav, srl, srlv,
    div, divu, mult, multu, mfhi, mflo, mthi, mtlo,
    slt, sltu, slti, sltiu, beq, bgez, bgezal, bgtz, blez,
    bltzal, bltz, bne, j, jal, jalr, jr,
    lb, lbu, lh, lhu, lw, lwl, lwr, sb, sh, sw, swl, swr, rfe
    ```

  - pseudo-instructions (some mnemonics are not real instructions)

    ```
    abs, mul, mulo, mulou, neg, negu, not, rem, remu, rol, ror,
    li, seq, sge, sgeu, sgt, sgtu, sle, sleu, sne, b, beqz, bge,
    bgeu, bgt, bgtu, ble, bleu, blt, bltu, bnez, la, ld, ulh,
    ulhu, ulw, sd, ush, usw, move,syscall, break, nop
    ```

# Actual "Code"

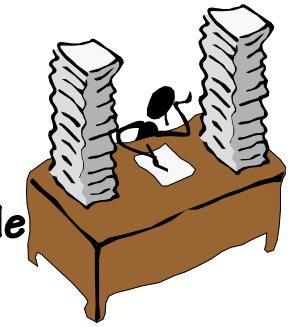
- Next we write ASSEMBLY *code* using the instruction mnemonics

A common convention, which originated with the 'C' programming language, is for the entry point (starting location) of a program to named "main".

```
.text
.globl main
main:
     add   $t0,$zero,$zero     # sum = 0
     add   $t1,$zero,$zero     # for (i = 0; ...
loop:
     addu  $t0,$t0,$t1    # sum = sum + i;
     addi  $t1,$t1,1      # for (...; ...; i++
     slti  $t2,$t1,5      # for (...; i<5;
     bne   $t2,$zero,loop
end: li    $v0, 10         # exit
     syscall
```

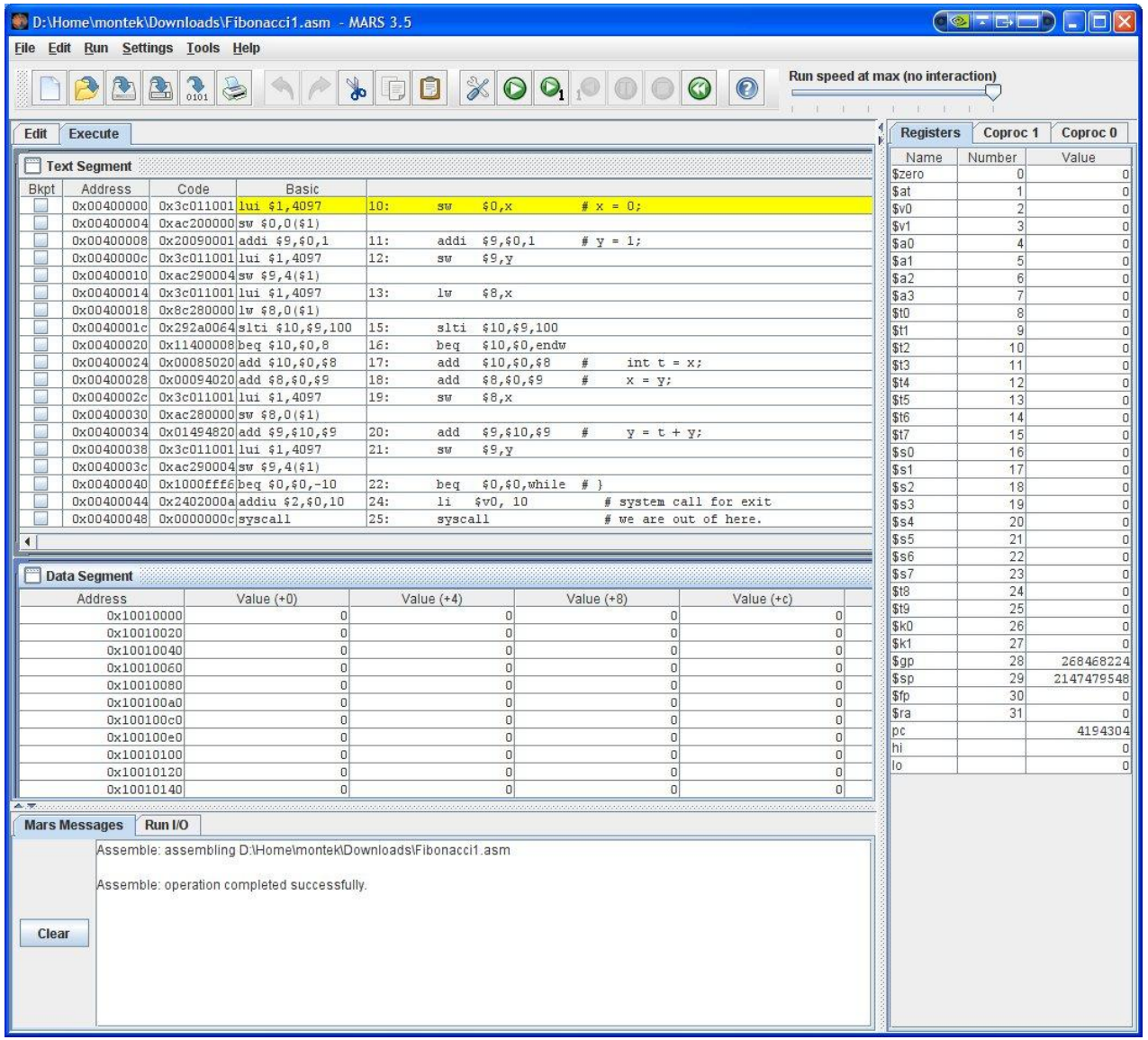
Bookkeeping:

1) Register $t0 is allocated as the "sum" variable
2) Register $t1 is allocated as the "i" variable



# MARS

- MIPS Assembler and Runtime Simulator
- Java application
- Runs on all platforms
- Links on class website
- (You'll need to download it for assignments)



# A Slightly More Challenging Program

- Add 5 numbers from a list ...
- sum = $n_0 + n_1 + n_2 + n_3 + n_4$
- In "C":

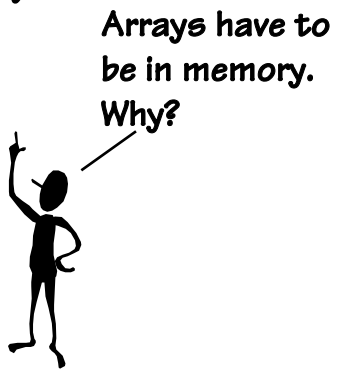
```
int i, sum;
int a[5] = {7,8,9,10,8};

main() {
     sum = 0;
     for (i=0; i<5; i++)
          sum = sum + a[i];
}
```

- Once more... let's encode it in assembly

# Variable Allocation

- We cheated in our last example. Generally, variables will be allocated to memory locations, rather than registers (Though clever optimization can often avoid it).
- This time we add the contents of an array

```
.data
sum:    .space 4
i:      .space 4
a:      .word 7,8,9,10,8
```




- ".word" allows us to initialize a list of sequential words in memory. The label represents the address of the first word in the list, or the name of the array

# The New Code

```
.text
.globl main
main:
     sw    $zero,sum      # sum = 0;
     sw    $zero,i        # for (i = 0;
     lw    $t1,i          # allocate register for i
     lw    $t0,sum        # allocate register for sum
loop:
     sll   $t2,$t1,2     # covert "i" to word offset
     lw    $t2,a($t2)    # load a[i]
     addu  $t0,$t0,$t2   # sum = sum + a[i];
     sw    $t0,sum       # update variable in memory
     addi  $t1,$t1,1     # for (...; ...; i++
     sw    $t1,i         # update memory
     slti  $t2,$t1,5     # for (...; i<5;
     bne   $t2,$zero,loop
end: li    $v0, 10       # exit
     syscall
```

# A Little "Weirdness"

Edit | Execute

**Text Segment**

| Bkpt | Address | Code | Basic | Source |
|---|---|---|---|---|
| | 0x00400000 | 0x3c011001 | lui $1,0x1001 | 8: sw $zero,sum # sum = 0; |
| | 0x00400004 | 0xac200000 | sw $0,0x0000($1) | |
| | 0x00400008 | 0x3c011001 | lui $1,0x1001 | 9: sw $zero,i # for (i = 0; |
| | 0x0040000c | 0xac200004 | sw $0,0x0004($1) | |
| | 0x00400010 | 0x3c011001 | lui $1,0x1001 | 10: lw $t1,i # allocate register for i |
| | 0x00400014 | 0x8c290004 | lw $9,0x0004($1) | |
| | 0x00400018 | 0x3c011001 | lui $1,0x1001 | 11: lw $t0,sum # allocate register for sum |
| | 0x0040001c | 0x8c280000 | lw $8,0x0000($1) | |
| | 0x00400020 | 0x00095080 | sll $10,$9,0x0002 | 13: sll $t2,$t1,2 # covert "i" to word offset |
| | 0x00400024 | 0x3c011001 | lui $1,0x1001 | 14: lw $t2,a($t2) # load a[i] |
| | 0x00400028 | 0x002a0821 | addu $1,$1,$10 | |
| | 0x0040002c | 0x8c2a0008 | lw $10,0x0008($1) | |
| | 0x00400030 | 0x010a4021 | addu $8,$8,$10 | 15: addu $t0,$t0,$t2 # sum = sum + a[i]; |
| | 0x00400034 | 0x3c011001 | lui $1,0x1001 | 16: sw $t0,sum # update variable in memory |
| | 0x00400038 | 0xac280000 | sw $8,0x0000($1) | |
| | 0x0040003c | 0x21290001 | addi $9,$9,0x0001 | 17: addi $t1,$t1,1 # for (...; ...; i++ |

**Data Segment**

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) | Value (+14) | Value (+18) | Value (+ |
|---|---|---|---|---|---|---|---|---|
| 0x10010000 | 0x00000000 | 0x00000000 | 0x00000007 | 0x00000008 | 0x00000009 | 0x0000000a | 0x00000008 | 0x0 |
| 0x10010020 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x10010040 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x10010060 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x10010080 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x100100a0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x100100c0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x100100e0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x10010100 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x10010120 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x10010140 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x10010160 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x10010180 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |
| 0x100101a0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x0 |

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

The Assembler rewrote some of our instructions. What's going on?

# A Coding Challenge

- What is the largest Fibonacci number less than 100?
  - Fibonacci numbers:

    $F_{i+1} = F_i + F_{i-1}$

    $F_0 = 0$

    $F_1 = 1$
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- In "C"

```
int x, y;

main() {
    x = 0;
    y = 1;
    while (y < 100) {
        int t = x;
        x = y;
        y = t + y;
    }
}
```

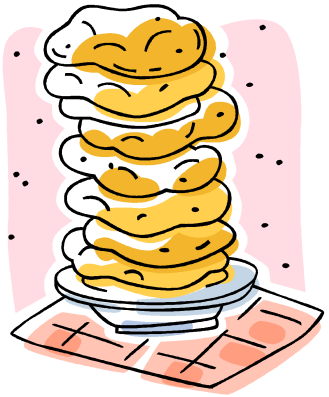
# MIPS Assembly Code

- In assembly

```
.data
.extern x 4
.extern y 4
x:    .space 4
y:    .space 4

.text
.globl main
main:
     sw    $zero,x            # x = 0;
     addi  $t1,$zero,1        # y = 1;
     sw    $t1,y
     lw    $t0,x
while:                        # while (y < 100) {
     slti  $t2,$t1,100
     beq   $t2,$zero,endw
     add   $t2,$zero,$t0      #     int t = x;
     add   $t0,$zero,$t1      #     x = y;
     sw    $t0,x
     add   $t1,$t2,$t1        #     y = t + y;
     sw    $t1,y
     beq   $zero,$zero,while  # }
end: li    $v0, 10            # exit
     syscall
```

# Next Time

- Parameterized Programs
- Procedures
- Stacks



- MIPS procedure linkage conventions