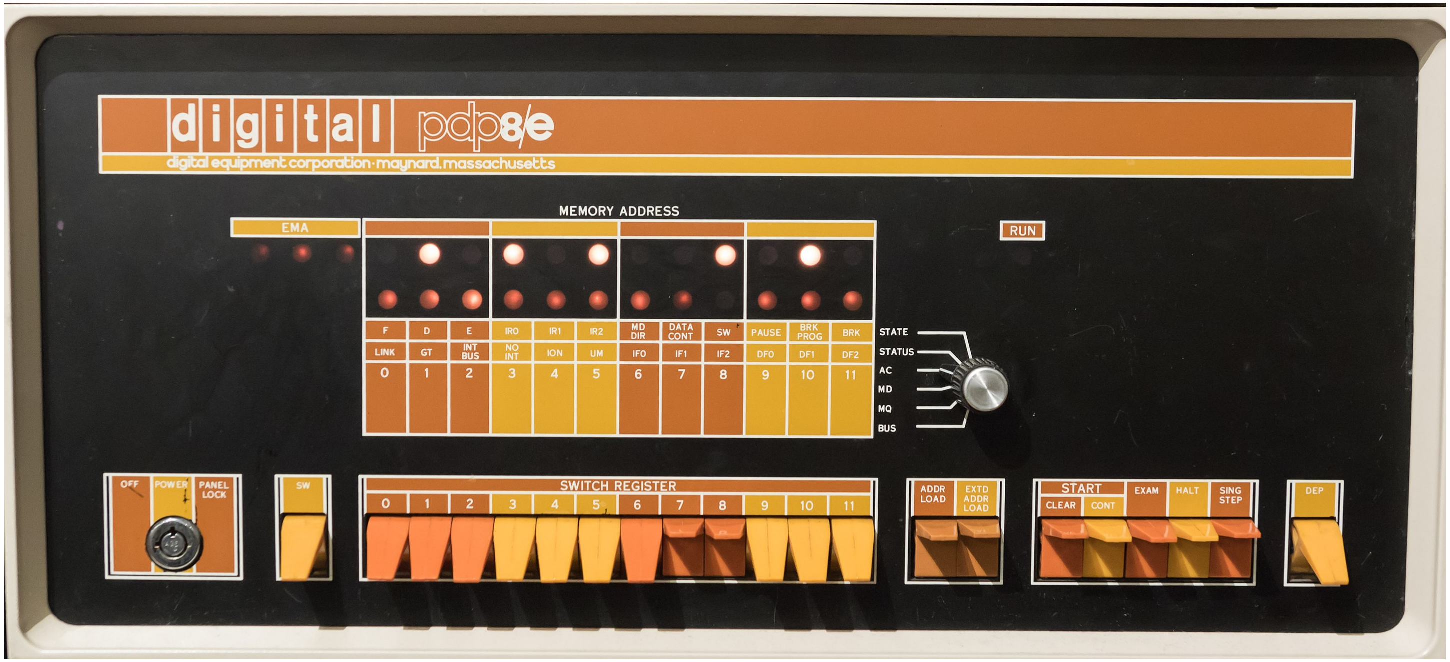
UMBC
AN HONORS UNIVERSITY

CMSC 411 – Lecture 9

Assembly & Simulation

Dr. Simsek



https://en.wikipedia.org/wiki/PDP-8/e

Today

Assembly programming

- structure of an assembly program
- assembler directives
- data and text segments
- allocating space for data

MIPS assembler: MARS

- development environment

A few coding examples

- self-study

Exercise: RGB-Gray Conversion

What is an Assembler?

A program for writing programs

Machine Language: 1's and 0's loaded into memory.

Assembly 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 → Binary Machine Language

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

STREAM of bits to be loaded into memory

Assembly: A Symbolic LANGUAGE for representing strings of bits
Assembler: 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 0x10010000`	Specifies address for start of data below
`.byte 1, 2, 3, 4`	Four byte values
`.byte 5, 6, 7, 8`	Another four byte values
`.word 1, 2, 3, 4`	Four word values (each is 4 bytes)

Resulting memory dump:

```
[0x10010000]    0x04030201  0x08070605  0x00000001  0x00000002
[0x10010010]    0x00000003  0x00000004  0x00000000  0x00000000
```

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

Assembly Source Language

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

`.data 0x10010000`	Specifies address for start of data below
`.byte 1, 2, 3, 4`	Four byte values
`.byte 5, 6, 7, 8`	Another four byte values
`.word 1, 2, 3, 4`	Four word values (each is 4 bytes)
`.asciiz "Comp 411"`	A zero (NULL) terminated ASCII string
`.align 2`	Align to next multiple of 2^2
`.word 0xfeedbeef`	A hex-encoded word value
`.text 0x00003000`	Specifies address for start of program text

space

Comp 411 ➔ 436f6d70 20343131

```
[0x10010000]    0x04030201  0x08070605  0x00000001  0x00000002
[0x10010010]    0x00000003  0x00000004  0x706d6f43  0x31313420
[0x10010020]    0x00000000  0xfeedbeef  0x00000000  0x00000000
```

These 0s are here because of asciiz!
ASCIIZ means that the string is terminated by the \0 (ASCII code 0) NUL character.

Change n in `.align n` and see what happens in the memory

Assembler Syntax

Assembler DIRECTIVES = Keywords prefixed with '.'

- 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 can be for data as well as for instructions
- Syntax: <start_of_line><label><colon>

```
.data
item:     .word 1                   # a data word

.text
start:    add     $3, $4, $2        # an instruction label
          sll     $3, $3, 8
          andi    $3, $3, 0xff
          beq     ..., ..., start
```

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`

  - not all implemented in all MIPS versions

- *Pseudo-instructions* (mnemonics that are not 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`
  - not real MIPS instructions; broken down by assembler into real ones

A Simple Programming Task

Add the numbers 0 to 4 …

$$0 + 1 + 2 + 3 + 4 = 10$$

Program in "C":

```
int i, sum;

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

Now let's code it in ASSEMBLY

Assembly Code: Sum.asm

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

$8 will have `sum`
$9 will have `i`

```
.text
main:
     add    $8,$0,$0       # sum = 0
     add    $9,$0,$0       # for (i = 0; ...
loop:
     add    $8,$8,$9       # sum = sum + i;
     addi   $9,$9,1        # for (...; ...; i++
     slti   $10,$9,5       # for (...; i<5;
     bne    $10,$0,loop
end:  ...                  # need something here to stop!
```

Bookkeeping:

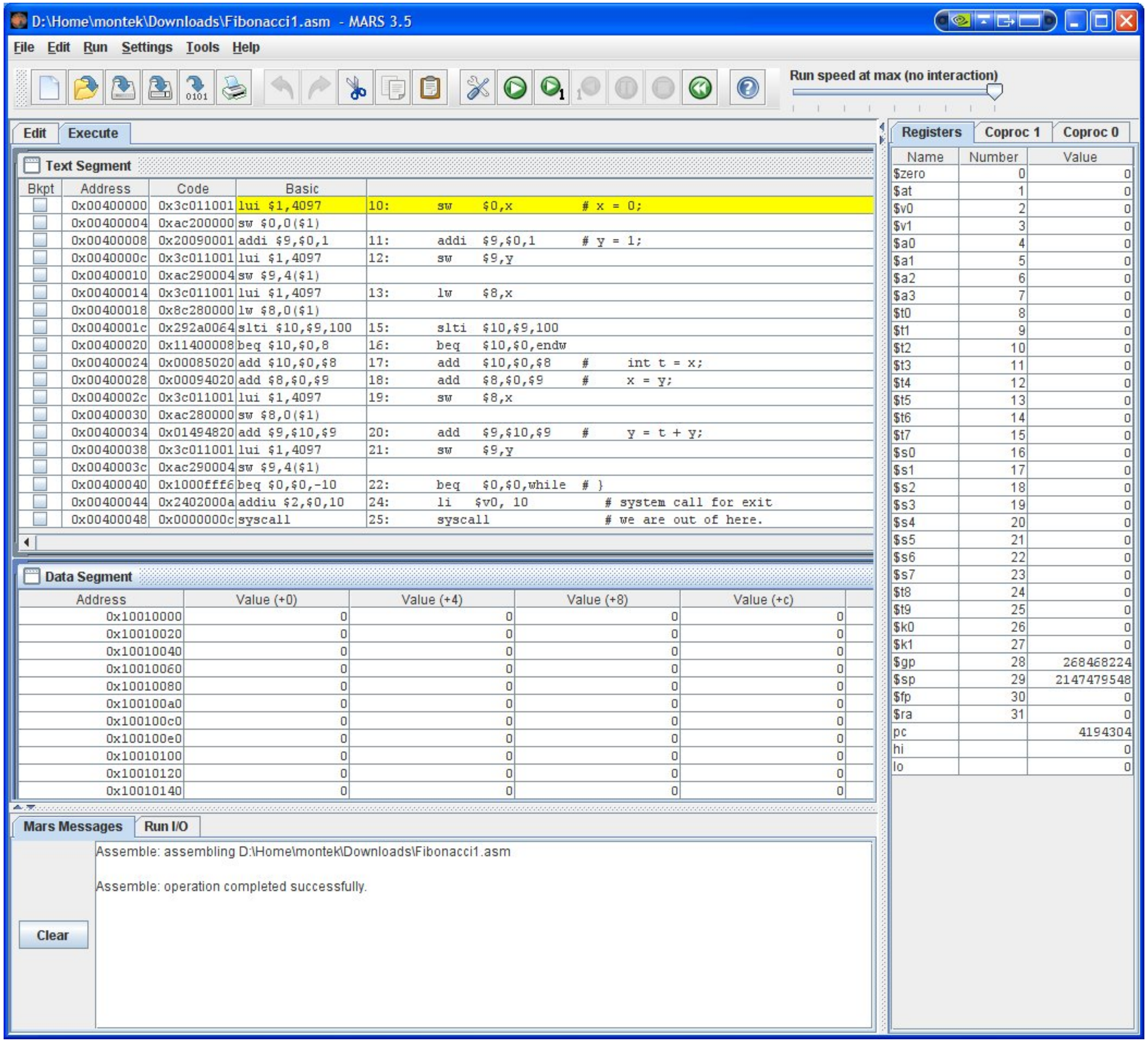
1) Register $8 is allocated as the "sum" variable
2) Register $9 is allocated as the "i" variable

We will talk about how to **exit** a program later

MARS

MIPS Assembler and Runtime Simulator (MARS)

- Java application
- Runs on all platforms
- Links on class website
- Download it now!



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 code it in assembly

Variable Allocation

Let's put variables in memory locations…

- … rather than registers

This time we add the contents of an array

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

Note: ".word" also works for an array of words

- allows us to initialize a list of sequential words in memory
- label represents the address of the first word in the list, or the name of the array
  - does this remind you of how C treats arrays as pointers?!

Note: ".space 4" means 4 bytes uninitialized

- ".word" needs initial value

The New Code: SumArray.asm

Note the small changes:

```
    .text
main:
        sw      $0,sum($0)      # sum = 0;
        sw      $0,i($0)        # for (i = 0;
        lw      $9,i($0)        # bring i into $9
        lw      $8,sum($0)      # bring sum into $8
loop:
        sll     $10,$9,2        # covert "i" to word offset
        lw      $10,a($10)      # load a[i]
        add     $8,$8,$10       # sum = sum + a[i];
        sw      $8,sum($0)      # update sum in memory
        addi    $9,$9,1         # for (...; ...; i++
        sw      $9,i($0)        # update i in memory
        slti    $10,$9,5        # for (...; i<5;
        bne     $10,$0,loop
end:    ...                     # code for exit here
```

A couple of shortcuts

Can skip the immediate or register field of lw/sw

- assumed to be zero
  - `lw $8,sum` ... is the same as ... `lw $8,sum($0)`
  - `lw $8,($10)` ... is the same as ... `lw $8,0($10)`
- assembler will fill in for you

A couple of shortcuts

Also, we can optimize code by eliminating intermediate updates in memory

- a good C compiler will do that automatically for you

```
main:
      add    $9,$0,$0      # i in $9 = 0
      add    $8,$0,$0      # sum in $8 = 0
loop:
      sll    $10,$9,2      # covert "i" to word offset
      lw     $10,a($10)    # load a[i]
      add    $8,$8,$10     # sum = sum + a[i];
      addi   $9,$9,1       # for (...; ...; i++
      slti   $10,$9,5      # for (...; i<5;
      bne    $10,$0,loop
      sw     $8,sum($0)    # update final sum in memory
      sw     $9,i($0)      # update final i in memory
end:   ...                 # code for exit here
```

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: Fibonacci.asm

In assembly

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

.text
main:
       sw     $0,x             # x = 0;
       addi   $9,$0,1          # y = 1;
       sw     $9,y
       lw     $8,x
while:                         # while (y < 100) {
       slti   $10,$9,100
       beq    $10,$0,endw
       add    $10,$0,$8        #     int t = x;
       add    $8,$0,$9         #     x = y;
       sw     $8,x
       add    $9,$10,$9        #     y = t + y;
       sw     $9,y
       j      while            # }
endw:
                               # code for exit here
                               # answer is in x
```