CSUN | CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

COMP 122

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Rev 8-27-23

ASSEMBLY Programming

Labs

Dr Jeff Drobman

website → drjeffsoftware.com/classroom.html

email → jeffrey.drobman@csun.edu

Lab Programs

1. "Hello World": I/O in MIPS & ARM
2. Number systems and radix conversion
3. BCD on LED
4. Moving data (memory <-> GR< -> FPU <-> CP0)
5. "Hello World" extended: loops, macros, functions/subroutines
6. Computation 1: Fibonacci (add, overflow)
7. Computation 2: Factorials (mult, overflow)
8. Bit-wise operations (bit masks, shifts); example: tic-tac-toe
9. Interrupt/Exception handler

10. Project 1: LED (MMIO, delay loops, speed slider)
11. Project 2: ISA design (logic design & sim new instructions)

2	3	3	3	3	3	3	5	5	7	8
LAB 1	LAB 2	LAB 3	LAB 4	LAB 5	LAB 6	LAB 7	LAB 8	LAB 9	Proj 1	Proj 2

Index

Lab

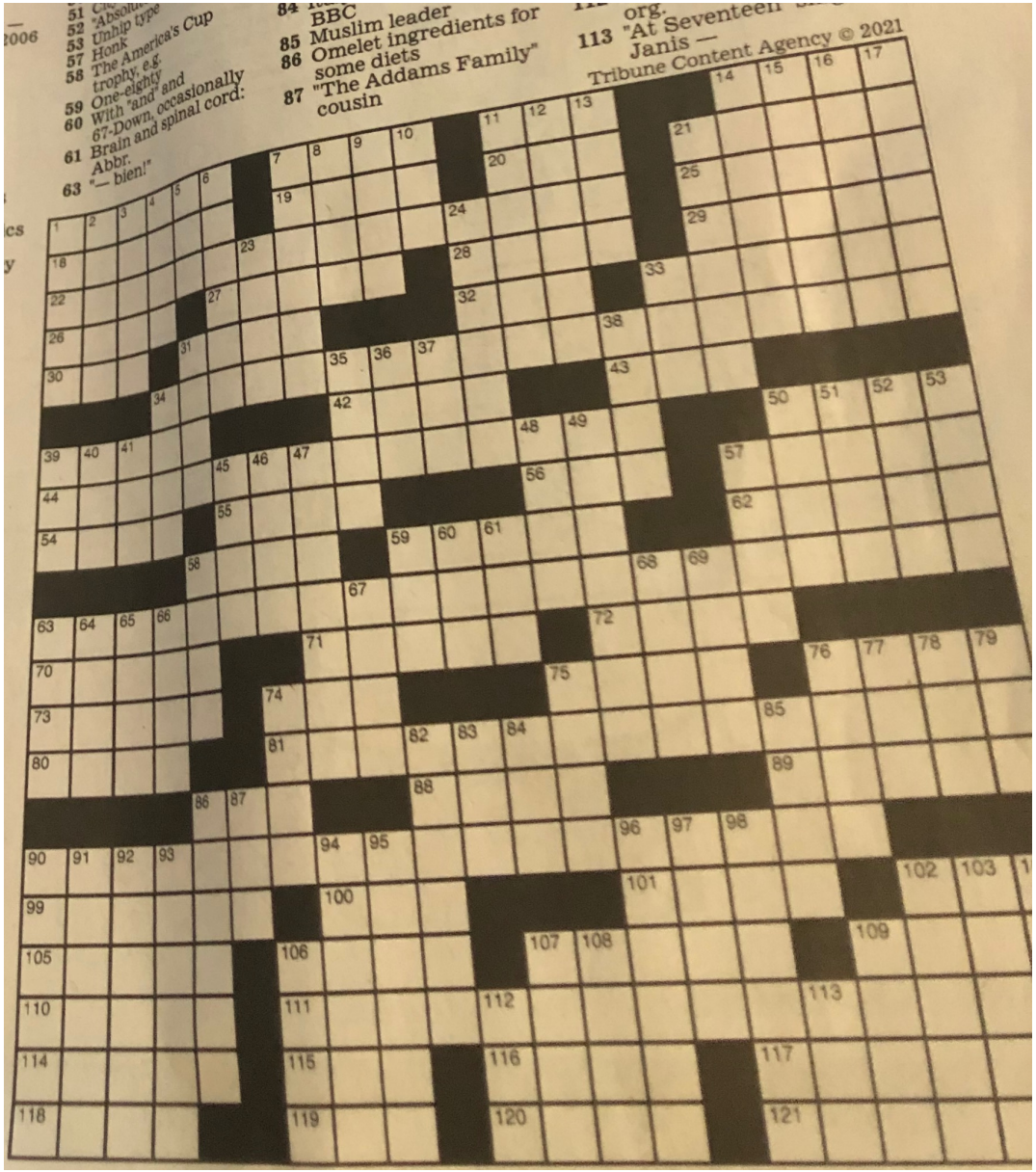
Assembly

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

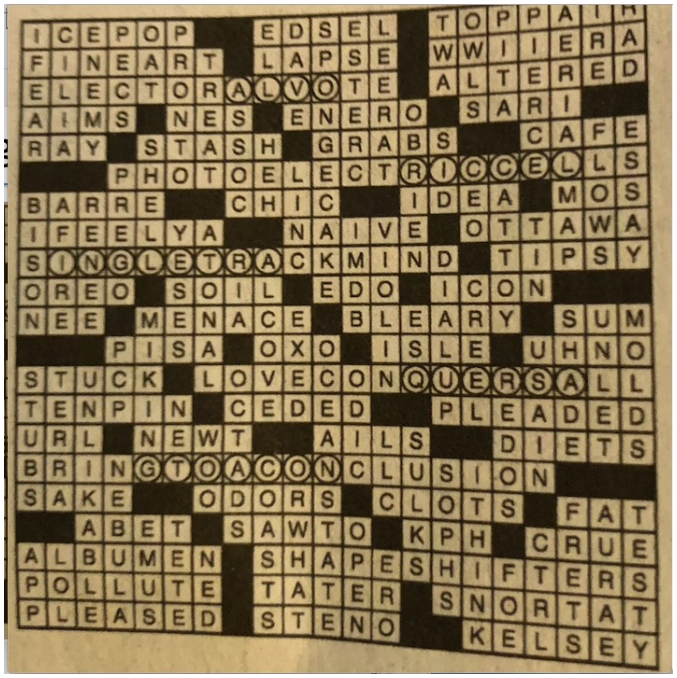
COMP122

Programming as a Puzzle

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2023



Given Solution



CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Baseline Instruction Set

Rev Jan 2021

Computation

- ALU
  - ADD
  - SUB
  - AND
  - OR
  - XOR
  - NOT
- MULT/DIV [opt]
- BIT
  - SET/CLR
  - TEST
- COMPARE
  - CMP
- SHIFT
  - SHIFT (A, L)
  - ROTATE

Memory

- Reg-Reg
  - MOV
- Reg-Mem
  - LOAD
  - STORE
  - MOV
- Mem-Mem
  - MOV
- Stack
  - PUSH
  - POP

Program Control

- JUMP
  - JUMP/GOTO
- BRANCH
  - BRA
  - BRCC
  - LOOP
- CALL
  - CALL/CALR/JAL
  - RET/RETFIE
- NOP

I/O

- I/O
  - IN
  - OUT
- Mem Mapped
  - MOV PORT
  - LOAD/STORE

System Control

- Reset
  - RESET
- Power
  - SLEEP/HALT

Data Types: Java/C → MIPS

.data *Static data segment*

```
//numeric
int final x = 3;  → .eqv  x, 3 //immutable
int x = 3;  →  x:  .word  3
short y = 5;  → y:  .half  5
byte b = 2;  →  x:  .byte  2
float f = 45.0;  →  f:  .float  45
double d = 55.0;  →  d:  .double  55
//non-numeric
char ch = 'a'; →  ch:  .ascii  "a"
String s = "hello" →  s:  .asciiz  "hello"
```

CPU Registers

- MIPS
- ARM

16-32 **GR**'s

- Includes specials
  - Stack: SP, FP
  - Globals: GP
  - Zero
- 64-bit
  - *Paired* 32-bit

- x86
  - Intel
  - AMD

Dedicated Registers

- 4 Accumulators
  - A, B, C, D
- 4 Pointers
  - SI, DI
  - BP, SP
- 5 Memory Segments
  - CS, DS, ES, FS, GS, SS
- 64-bit
  - *Telescoping*

➢ **PC** dedicated

AH	AL

AX

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

MARS

DR JEFF SOFTWARE
INDIE APP DEVELOPER

MARS (MIPS Assembler and Runtime Simulator)

Registers

Mars

MARS 4.5

File Edit Run Settings Tools Help

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff11
$13 (cause)	13	0x00000000
$14 (epc)	14	0x00000000

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Registers | Coproc

Name	Float
$f0	0x00000000
$f1	0x00000000
$f2	0x00000000
$f3	0x00000000
$f4	0x00000000
$f5	0x00000000
$f6	0x00000000
$f7	0x00000000
$f8	0x00000000
$f9	0x00000000
$f10	0x00000000

GP Registers

Register use convention: Hennessy & Patterson

Figure 2.8.1: What is and what is not preserved across a procedure call (COD Figure 2.11).

If the software relies on the frame pointer register or on the global pointer register, discussed in the following subs
preserved.

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

- $a(0:3) *args*
- $at, $k(0:1) *reserved*
- $v(0:1) *values*
- $t(0-9) *temp*
- $s(0:7) *saved*
- $gp *global ptr*
- $sp *stack ptr*
- $fp *frame ptr*
- $ra *return addr*

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER

arm Developer IP PRODUCTS TOOLS AND SOFTWARE ARCHITECTURES INTERNET OF THINGS COMMUNITY SUPPORT DOCUMENTATION DOWNLOADS

Running Software on Your Target

Transfer your executable image to a target device.

Learn more

Raspberry Pi 4

Writing Arm Assembly Code

Learn Arm assembly language with our curated resources.

Learn more

ARMsim

Developing Embedded Software

Building Your First Embedded Image

Retargeting Output to UART

Creating an Event-Driven Embedded Image

Event driven

Assembly

Learning about the instruction set

Assembly instructions are the fundamental building blocks of any program. If you are going to write assembly code, you will need to understand what instructions are available to you.

The precise set of available instructions for a particular device is called the instruction set. The Arm architecture supports three *Instruction Set Architectures* (ISAs): A64, A32 and T32. Use these resources for more information:

- Learn more about the different instruction sets supported by the Arm architecture.
- Use the ISA exploration tools to discover the available A64, A32, and T32 instructions in easy-to-browse XML and HTML formats.
- Refer to the Arm Architecture Reference Manuals (A-profile, R-profile, and M-profile) for definitive ISA details.

Assembly

Learning about assembly language

Although the instruction set reference materials described in the Overview are comprehensive, they do not provide the best starting point for beginners.

The following resources introduce the basic concepts of programming in Arm assembly language:

- The Cortex-A Series Programmer's Guide explains architectural fundamentals and an introduction to assembly language code, along with other useful information for programmers.
- Arm Assembly Language: Fundamentals and Techniques by William Hohl is a popular resource with the Arm University Program. This book discusses the basics of assembly language.
- Embedded Systems Fundamentals with Arm Cortex-M based Microcontrollers: A Practical Approach by Dr Alexander G. Dean includes a chapter correlating C programming features with those in assembly code.

The Arm Compiler 5 toolchain (executable name `armasm`) uses a different syntax for assembly code to Arm Compiler 6 (executable name `armclang`) and GNU (executable name `as`). Although the instructions are mostly the same regardless of toolchain, the syntax around the instructions varies.

Hello World in Assembly

Writing your first assembly program

Here are some examples that you can follow to get started with Arm assembly language.

- "Hello World" in Assembly is an Arm Community blog post that shows how to build a simple Arm assembly program with GCC, running either natively or with a cross-compiler.
- Assembling armasm and GNU syntax assembly code in the Arm Compiler Software Development Guide demonstrates another Arm assembly program, this time using the Arm Compiler 6 toolchain.
- Using the integrated assembler in the Arm Compiler User Guide provides another example.

Embedding Assembly in C

Mixing C, C++, and assembly code

Even though you can now write Arm assembly code, you probably do not want to use it to hand-code your entire application.

You will probably want to write a small number of key functions in assembly, and call those functions from your main application code. You might want to do this to make use of existing assembly code, but the rest of your project is in C or C++.

- Calling assembly functions from C and C++ in the Arm Compiler User Guide shows you how to make function calls from C/C++ code to assembly code using the Arm Compiler 6 toolchain.
- Beyond Hello World: Advanced Arm Compiler 5 Features provides a similar example using Arm Compiler 5 within the DS-5 IDE.

3-Level Example

C

```
    #include <stdio.h>
int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;
    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

Assembly--MIPS

```
addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4, 32($29)
sw       $5, 36($29)
sw       $0, 24($29)
sw       $0, 28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu    $14, $14
addiu    $8, $14, 1
slti     $1, $8, 101
sw $8,   28($29)
mflo     $15
addu     $25, $24, $15
bne      $1, $0, -9
sw       $25, 24($29)
lui      $4, 4096
lw       $5, 24($29)
jal      1048812
addiu    $4, $4, 1072
lw       $31, 20($29)
addiu    $29, $29, 32
jr       $31
move     $2, $0
```

Figure 7.1.2: MIPS machine language code for a routine to compute and print the sum of the squares of integers between 0 and 100 (COD Figure A.1.2).

Machine -- binary

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

Assembly Example

PIC18F

```
MPASM  5.37                    LAB4.ASM   10-12-2015  21:21:12        PAGE  1


LOC  OBJECT CODE     LINE SOURCE TEXT
  VALUE

                     00001 #include <p18F458.inc>
                     00001         LIST
                     00002 ; P18F458.INC  Standard Header File, Version 1.10   Microchip T
Message[301]: MESSAGE: (Processor-header file mismatch.  Verify selected processor.)
                     01714         LIST
                     00002 #include <lab4.inc>
  00000100           00001 START EQU 0X100;address 256
000000               00002         org 0
000000 EF84 F000     00003         GOTO _start
000008               00004         org 0x08
000008 0010          00005         RETFIE
000018               00006         org 0x018
000018 0010          00007         RETFIE
                     00008
000100               00009         org START
000100 0000          00010 ZERO DB 0
000102 0001          00011 ONE DB 1
000104 000A          00012 TEN DB D'10'
000106 00FF          00013 ALL1 DB 0xFF
000108 0000          00014 _start NOP
                     00015
                     00016
                     00003
00010A EF85 F000     00004 loop GOTO loop
                     00005         end
MPASM  5.37                    LAB4.ASM   10-12-2015  21:21:12        PAGE  2
```

Space for code

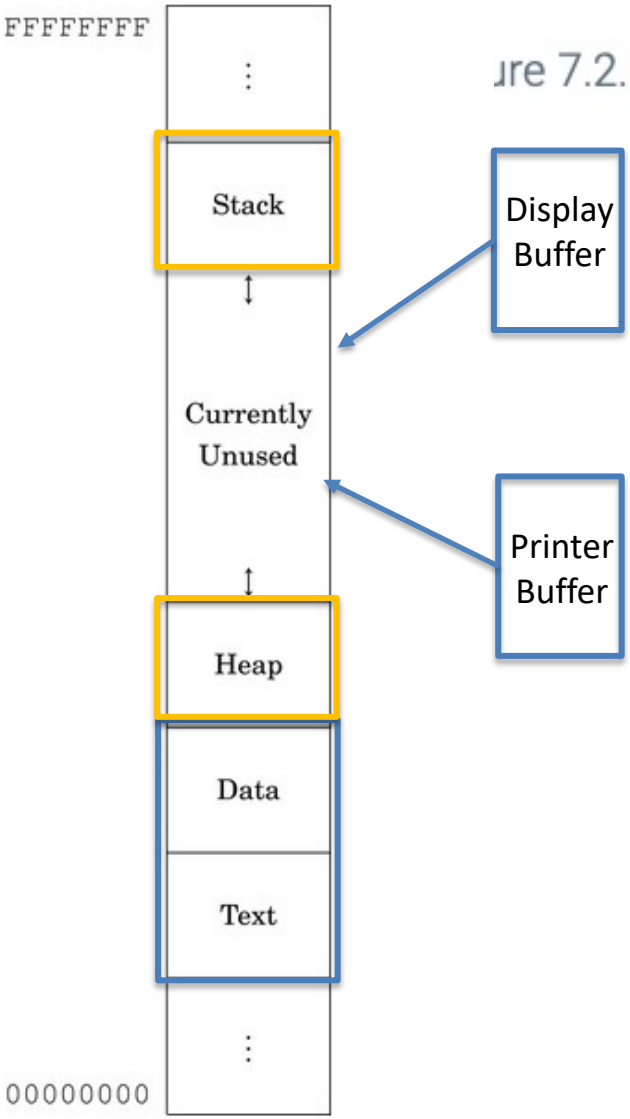
```
        MOVLW   D'10'
        MOVWF   COUNTER
        LFSR    1,DATAPTR

LOOP    CLRF POSTINC1
        DECF COUNTER
        BNZ LOOP

loop GOTO loop
     end
```

Memory Segments

MIPS



Display Buffer

Printer Buffer

ure 7.2.1: Object file (COD Figure A.2.1).

A UNIX assembler produces an object file with six distinct sections.

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information

Typical memory layout for a program with a 32-bit address space.

MARS

Memory Map

MARS (MIPS Assembler and Runtime Simulator)

MIPS Memory Configuration

0xffffffff	memory map limit address
0xffffffff	kernel space high address
0xffff0000	MMIO base address
0xfffeffff	kernel data segment limit address
0x90000000	.kdata base address
0x8ffffffc	kernel text limit address
0x80000180	exception handler address
0x80000000	kernel space base address
0x80000000	.ktext base address
0x7fffffff	user space high address
0x7fffffff	data segment limit address
0x7ffffffc	stack base address
0x7fffeffc	stack pointer $sp
0x10040000	stack limit address
0x10040000	heap base address
0x10010000	.data base address
0x10008000	global pointer $gp
0x10000000	data segment base address
0x10000000	.extern base address
0x0ffffffc	text limit address
0x00400000	.text base address

Configuration
- (●) Default
- ( ) Compact, Data at Address 0
- ( ) Compact, Text at Address 0

System Call (*syscall*)

MIPS

Figure 7.9.1: System services (COD Figure A.9.1).

SPIM

Service	System call code	Arguments	Result
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, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

System Call (*syscall*)

SPIM

Services 1 through 17 are compatible with the SPIM simulator, other than Open File (13) as described in the Notes below the table. Services 30 and higher are exclusive to MARS.

50-59 are GUI boxes

MessageDialog	55	$a0 = address of null-terminated string that is the message to user $a1 = the type of message to be displayed: 0: error message, indicated by Error icon 1: information message, indicated by Information icon 2: warning message, indicated by Warning icon 3: question message, indicated by Question icon other: plain message (no icon displayed)

System Call (*syscall*)

SPIM provides a small set of operating system—like services through the system call
program loads the system call code (see the figure above) into register **$v0** and argu
point values). System calls that return values put their results in register **$v0** (or **$f0**
code prints **"the answer = 5"**:

```
    .data
str:
    .asciiz    "the answer = "
    .text

    li         $v0, 4          # system call code for print_str
    la         $a0, str        # address of string to print
    syscall                    # print the string

    li         $v0, 1          # system call code for print_int
    li         $a0, 5          # integer to print
    syscall                    # print it
```

Load address

The **print_int** system call is passed an integer and prints it on the console. **print**

Macros with C *printf*

As an example, suppose that a programmer needs to print many numbers. The library routine **printf** accepts a format string and one or more values to print as its arguments. A programmer could print the integer in register **$7** with the following instructions:

```
        .data
int_str: .asciiz "%d"
        .text
        la $a0, int_str # Load string address
                        # into first arg
        mov $a1, $7     # Load value into
                        # second arg
        jal printf      # Call the printf routine
```

Embedded format specifier for "printf"

The **.data** directive tells the assembler to store the string in the program's data segment, and the **.text** directive tells the assembler to store the instructions in its text segment.

However, printing many numbers in this fashion is tedious and produces a verbose program that is difficult to understand. An alternative is to introduce a macro, **print_int**, to print an integer:

```
        .data
int_str: .asciiz "%d"
        .text
        .macro print_int($arg)
        la $a0, int_str # Load string address into
                        # first arg
        mov $a1, $arg   # Load macro's parameter
                        # ($arg) into second arg
        jal printf      # Call the printf routine
        .end_macro
print_int($7)
```

.macro

Macros with C *printf*

The macro has a *formal parameter*, `$arg`, that names the argument to the macro. When the macro is expanded, the argument from a call is substituted for the formal parameter throughout the macro's body. Then the assembler replaces the call with the macro's newly expanded body. In the first call on `print_int`, the argument is `$7`, so the macro expands to the code

```
la   $a0, int_str
mov  $a1, $7
jal  printf
```

In a second call on `print_int`, say, `print_int($t0)`, the argument is `$t0`, so the macro expands to

```
la   $a0, int_str
mov  $a1, $t0
jal  printf
```

What does the call `print_int($a0)` expand to?

Answer

```
la   $a0, int_str
mov  $a1, $a0
jal  printf
```

Lab

LAB 1

Hello World

MIPS

Comparison: "Hello World"

Lab 1

C

```
#include <stdio.h>
void main (void) {
  printf("Hello world!\n");
}
```

C++

```
#include <iostream>
void main () {
  std::cout << "Hello world!\n";
}
```

Java

```
public class helloWorld {
public static void main (String[] args) {
  system.out.println ("Hello world!");
} }
```

Javascript

```
//myfile.js
Console.log("Hello world!");
```

Python

```
Print "Hello world!"
```

Comparison: "Hello World"

Basic

```
10 PRINT "Hello, world!"
20 END
```

note: line numbers!

VB

```
Public Sub Main()
  MsgBox "Hello, world!"
End Sub
```

OOP + **GUI**

C#

```
using System;

internal static class HelloWorld
{
    private static void Main()
    {
        Console.WriteLine("Hello, world!");
    }
}
```

OOP + console

DOS

```
@echo Hello World!
```

script (for console)

Comparison: "Hello World"

- all console

PHP

```php
<?php
print "Hello world!";
?>
```

x86 Assembly

```
.model small
.stack 100h

.data
msg     db      'Hello world!$'

.code
start:
        mov     ah, 09h
        lea     dx, msg
        int     21h
        mov     ax, 4C00h ;
        int     21h
end start
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Hello World in Java

DR JEFF SOFTWARE
INDIE APP DEVELOPER


```java
/* CSUN COMP110 header
student: Jeff Drobman
ver date: 2-4-21
file:  Hello.java
Lab 1:  just Hello
*/
//imports
import javax.swing.*;
//main class
public class Hello.java {
//static global DATA
   static String hello = "Hello World!";
//main method
   public static void main(String[] args) {
      //OUTPUT
      System.out.println(hello);//console
      JOptionPane.showMessageDialog(null, hello);//GUI box
   }//end main method
}//end class
```

Hello World in C

Standard C Implementation

A traditional introduction to many languages is the "Hello World" program. In C, this looks something like this:

```
#include <stdio.h>

int main(void) {
  printf("Hello, world.\n");
  return 0;
}
```

Hello World in C++

HelloWorld.cpp

```
//===========================================================
// Name        : HelloWorld.cpp
// Author      :
// Version     :
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//===========================================================

#include <iostream>
using namespace std;

int main() {
    cout << "!!!Hello World!!!" << endl; // prints !!!
    return 0;
}
```

JHD-HelloWo... | makefile | objects.mk | sources.mk

```
1  //===========================================================
   // Name        : JHD-HelloWorld.cpp
   // Author      : JHD
4  // Version     :
5  // Copyright   : copyright JHD
6  // Description : Hello World in C++, Ansi-style
7  //===========================================================
8
9  #include <iostream>
10 using namespace std;
11
12 int main() {
13     cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
14     return 0;
15 }
16
```

Hello World C++ Extended


```
#include <iostream>
using namespace std;

int main () {
    // Say HelloWorld five times
    for (int index = 0; index < 5; ++index)
      cout << "HelloWorld!" << endl;
    char input = 'i';
    cout << "To exit, press 'm' then the 'Enter' key." << endl;
    cin  >> input;
    while(input != 'm') {
        cout << "You just entered '" << input << "'. "
             << "You need to enter 'm' to exit." << endl;
        cin  >> input;
    }
    cout << "Thank you. Exiting." << endl;
    return 0;
}
```

Hello World in Python


```
>>> print "Hello World!"
Hello World!
>>>
```

```
Last login: Fri Aug 16 13:38:24 on ttys000
Jeffreys-MacBook-Air:~ jhdphd$ python
Python 2.7.10 (default, Aug 17 2018, 17:41:52)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World!"
Hello World!
>>>
```

```
A. HISTORY OF THE SOFTWARE
==========

Python was created in the early 1990s by Guido van Rossum at Stichting
Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands
as a successor of a language called ABC.  Guido remains Python's
principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for
National Research Initiatives (CNRI, see http://www.cnri.reston.va.us)
in Reston, Virginia where he released several versions of the
software.

In May 2000, Guido and the Python core development team moved to
BeOpen.com to form the BeOpen PythonLabs team.  In October of the same
year, the PythonLabs team moved to Digital Creations (now Zope
Corporation, see http://www.zope.com).  In 2001, the Python Software
Foundation (PSF, see http://www.python.org/psf/) was formed, a
non-profit organization created specifically to own Python-related
Intellectual Property.  Zope Corporation is a sponsoring member of
the PSF.

All Python releases are Open Source (see http://www.opensource.org for
Hit Return for more, or q (and Return) to quit:
```

Python

ASCII

Lab 1

Binary, hexadecimal, and decimal equivalents for each character in "Hello World"

Character	Binary	Hexadecimal	Decimal
H	01001000	48	72
e	01100101	65	101
l	01101100	6C	108
l	01101100	6C	108
o	01101111	6F	111
	00100000	20	32
W	01010111	57	87
o	01101111	6F	111
r	01110010	62	98
l	01101100	6C	108
d	01100100	64	100
NUL	00000000	00	0

Assembler *Endian*

est address. Some processors, such as the ARM, can be configured as either little-endian or big-endian. The Linux operating system, by default, configures the ARM processor to run in little-endian mode .

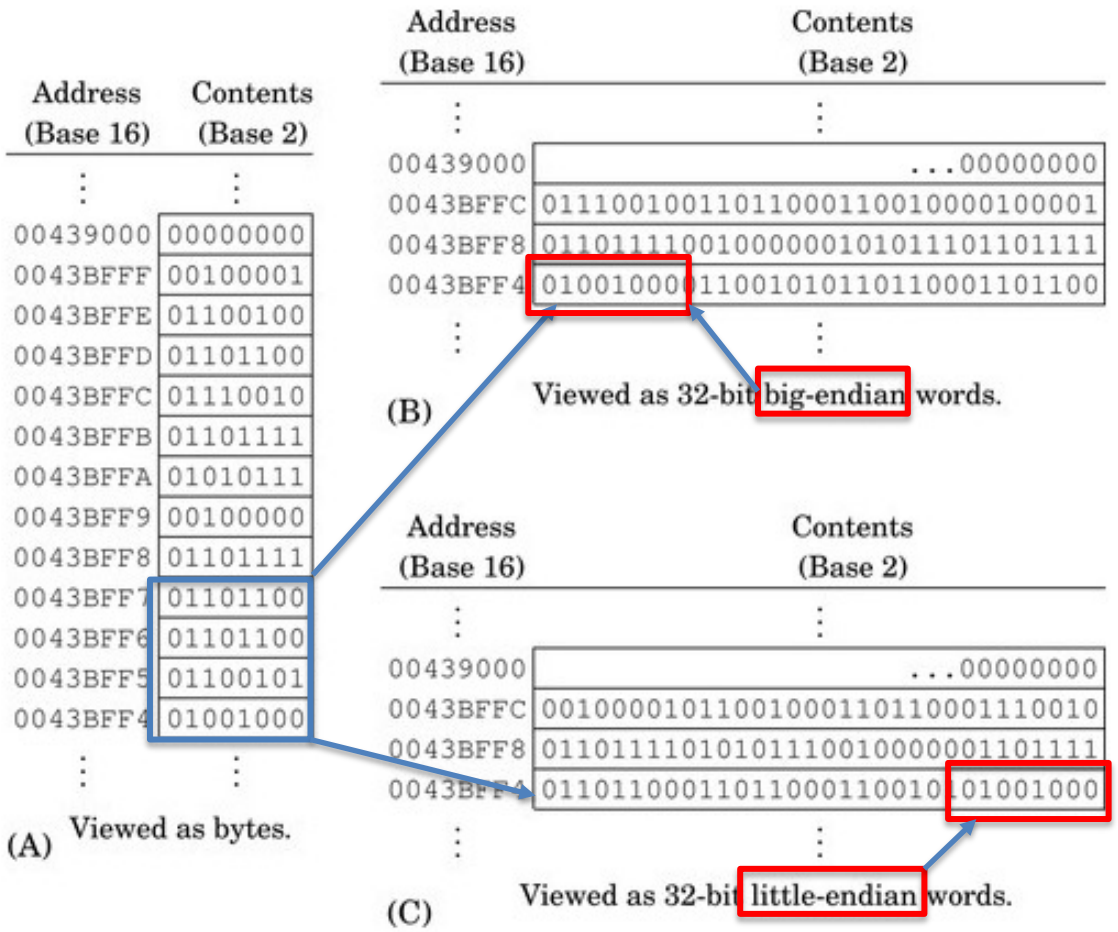


FIGURE 1.6 A section of memory.

Lab

LAB 1A

Hello World

MIPS

Lab 1A

- Store "Hello World!"
  - into *data* seg
  - copy from *data* into *heap* seg
- Print "Hello World!"
  - Console
  - GUI box

Java version of Lab 1A

OUTPUT

```
 ----jGRASP exec: java Lab1Hello122
Hello World!

 ----jGRASP: operation complete.
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Java version of Lab 1A

Header

CODE

```
/* CSUN COMP110 header
student: Jeff Drobman
ver date: 1-16-21
file:  Lab1Hello122.java
Lab 1A:  Hello basic
*/
//imports
import java.util.*;
import javax.swing.*;
//main class
public class Lab1Hello122 {
//static global DATA
   static String hello = "Hello World!";
//main method
   public static void main(String[] args) {
      //simulate a "heap" = dynamic DATA segment
      char[] heap = new char[1000]; //reserve 1000 bytes
      char[] charHello = hello.toCharArray();

      //copy to heap (all words)
      for(int i=0; i<hello.length(); i++)
         heap[i] = charHello[i];

      //OUTPUT
      System.out.println(hello);//console
      JOptionPane.showMessageDialog(null, hello);//GUI box

   }//end main method
```

Static data

Copy to heap

Output

Memory "Move"

Address	Value (+0)	Value (+4)	Value (+
0x10010000	l l e H	o W o	\0
0x10010020	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x10010040	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x10010060	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x100100c0	\0 \0 \0 \0	\0 \0 \0 \0	\0

0x10010000 (.data)

Load

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10040000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x10040000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x6666654a
$t2	10	0x10010018
$t3	11	0x1004000c

Store

Address	Value (+0)	Value (+4)	Value (-
0x10040000	l l e H	o W o	\0
0x10040020	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x10040040	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x10040060	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x10040080	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x100400a0	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x100400c0	\0 \0 \0 \0	\0 \0 \0 \0	\0

0x10040000 (heap)

MIPS Assembly

MIPS Lab 1A

mips-Hello.asm

```
1  ## Hello World
2  ## by Jeff Drobman
3  ##
4  .data
5  hello: .asciiz  "Hello World"
6  #
7  .text
8  lw $t1,hello
9  sw $t1,hello
10
```

Header

Static data

CODE

Load-Store

Assembled Code

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	8: lw $t1,hello
	0x00400004	0x8c290000	lw $9,0x00000000($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	9: sw $t1,hello
	0x0040000c	0xac290000	sw $9,0x00000000($1)	

MIPS Loads

EA

lw $t1,-100($t2)	Prim	Load word : Set $t1 to contents of effective memory word address
lw $t1,($t2)	*	Load Word : Set $t1 to contents of effective memory word address
lw $t1,-100		Load Word : Set $t1 to contents of effective memory word address
lw $t1,100		Load Word : Set $t1 to contents of effective memory word address
lw $t1,100000		Load Word : Set $t1 to contents of effective memory word address
lw $t1,100($t2)	*	Load Word : Set $t1 to contents of effective memory word address
lw $t1,100000($t2)		Load Word : Set $t1 to contents of effective memory word address
lw $t1,label		Load Word : Set $t1 to contents of memory word at label's address
lw $t1,label($t2)		Load Word : Set $t1 to contents of effective memory word address
lw $t1,label+100000		Load Word : Set $t1 to contents of effective memory word address
lw $t1,label+100000($t2)		Load Word : Set $t1 to contents of effective memory word address
li $t1,-100		Load Immediate : Set $t1 to 16-bit immediate (sign-extended)
li $t1,100		Load Immediate : Set $t1 to unsigned 16-bit immediate (zero-extended)
li $t1,100000		Load Immediate : Set $t1 to 32-bit immediate
la $t1,($t2)		Load Address : Set $t1 to contents of $t2
la $t1,-100		Load Address : Set $t1 to 16-bit immediate (sign-extended)
la $t1,100		Load Address : Set $t1 to 16-bit immediate (zero-extended)
la $t1,100000		Load Address : Set $t1 to 32-bit immediate
la $t1,100($t2)		Load Address : Set $t1 to sum (of $t2 and 16-bit immediate)
la $t1,100000($t2)		Load Address : Set $t1 to sum (of $t2 and 32-bit immediate)
la $t1,label		Load Address : Set $t1 to label's address
la $t1,label($t2)		Load Address : Set $t1 to sum (of $t2 and label's address)
la $t1,label+100000		Load Address : Set $t1 to sum (of label's address and 32-bit immediate)
la $t1,label+100000($t2)		Load Address : Set $t1 to sum (of label's address, 32-bit immediate, and $t2)

Assembled MIPS Loads

EA: address calculations

Address	Code	Basic	Source
0x00400000	0x24090004	addiu $9,$0,0x00000004	12: li $t1, 4 #next word
0x00400004	0x3c011001	lui $1,0x00001001	13: la $t2, abcd #address in .data
0x00400008	0x342a0000	ori $10,$1,0x00000000	
0x0040000c	0x3c011004	lui $1,0x00001004	14: li $t3, 0x10040020 #imm
0x00400010	0x342b0020	ori $11,$1,0x00000020	
0x00400014	0x3c011004	lui $1,0x00001004	15: la $t4, 0x10040020 #address
0x00400018	0x342c0020	ori $12,$1,0x00000020	
0x0040001c	0x3c0dffff	lui $13,0x0000ffff	16: lui $t5, 0xffff
0x00400020	0x35ad2a2a	ori $13,$13,0x00002a2a	17: ori $t5, $t5, 0x2a2a
0x00400024	0x8d4e0000	lw $14,0x00000000($10)	19: lw $t6, 0($t2) #aaaa
0x00400028	0x8d4f000c	lw $15,0x0000000c($10)	20: lw $t7, 12($t2) #dddd
0x0040002c	0x8d510000	lw $17,0x00000000($10)	22: lw $s1, ($t2) #aaaa
0x00400030	0x3c011001	lui $1,0x00001001	23: lw $s2, abcd+4 #bbbb
0x00400034	0x8c320004	lw $18,0x00000004($1)	
0x00400038	0x3c011001	lui $1,0x00001001	24: lw $s3, abcd+4($t1) #cccc
0x0040003c	0x00290821	addu $1,$1,$9	
0x00400040	0x8c330004	lw $19,0x00000004($1)	
0x00400044	0xad710000	sw $17,0x00000000($11)	26: sw $s1, ($t3) #aaaa
0x00400048	0xad720004	sw $18,0x00000004($11)	27: sw $s2, 4($t3) #dddd
0x0040004c	0x3c011001	lui $1,0x00001001	28: sw $s3, empty+4 #bbbb
0x00400050	0xac330018	sw $19,0x00000018($1)	

Assembled MIPS Loads

Address	Code	Basic	Source
0x00400054	0x3c011001	lui $1,0x00001001	29: sw $s4, empty+4($t1) #cccc
0x00400058	0x00290821	addu $1,$1,$9	
0x0040005c	0xac340018	sw $20,0x00000018($1)	

EA: address calculations

Address	Code	Basic	Source
0x00400058	0x00290821	addu $1,$1,$9	
0x0040005c	0xac340018	sw $20,0x00000018($1)	
0x00400060	0xad6c0008	sw $12,0x00000008($11)	30: sw $t4, 8($t3)
0x00400064	0x24090100	addiu $9,$0,0x00000100	32: li $t1, 256 #short I
0x00400068	0x240afffe	addiu $10,$0,0xffff...	33: li $t2, -2 #short neg
0x0040006c	0x3c011234	lui $1,0x00001234	34: li $t3, 0x12345678 #long I
0x00400070	0x342b5678	ori $11,$1,0x00005678	
0x00400074	0x3c011234	lui $1,0x00001234	35: la $t4, 0x12345678 #same?
0x00400078	0x342c5678	ori $12,$1,0x00005678	
0x0040007c	0x012a6820	add $13,$9,$10	36: add $t5,$t1,$t2 #R format
0x00400080	0x214d0ce4	addi $13,$10,0x0000...	37: addi $t5,$t2, 3300 #I format--too large?

MIPS Assembly

MIPS Lab 1A

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,0x00001001	8: lw $t1,hello
	0x00400004	0x8c290000	lw $9,0x00000000($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	9: sw $t1,hello
	0x0040000c	0xac290000	sw $9,0x00000000($1)	

Static data

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	l l e H	o W o	\0 d l r	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

$sp	29	0x7fffeffc

MIPS Assembly

MIPS | Lab 1A

```
## Hello World
## by Jeff Drobman
##
#register map:
#$t1=string data, $t2=heap pointer
.eqv heapHi, 0x1004
.data
#heap: 0x10040000
hello: .asciiz  "Hello World\n"
.text
#store word 1 in heap
lw $t1, hello
lui $t2, heapHi
sw $t1, ($t2)
#store word 2 in heap
lw $t1, hello+4
add $t2, $t2, 4
sw $t1, ($t2)
#print on console using "Syscall 4"
li $v0, 4 #print code=
la $a0, hello #address (pointer)
syscall
nop #extra
break 0 #System.exit(0)
```

Setup (init static data)

Code

Mars Messages | Run I/O

```
Hello World
```

MIPS Assembly

MIPS Lab 1A

Edit Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400014	0x8c290004	lw $9,0x00000004($1)	
	0x00400018	0x214a0004	addi $10,$10,0x0000...	17: add $t2, $t2, 4
	0x0040001c	0xad490000	sw $9,0x00000000($10)	18: sw $t1, ($t2)
	0x00400020	0x24020004	addiu $2,$0,0x00000004	20: li $v0, 4 #print code=
	0x00400024	0x3c011001	lui $1,0x00001001	21: la $a0, hello #address (pointer)
	0x00400028	0x34240000	ori $4,$1,0x00000000	
	0x0040002c	0x0000000c	syscall	22: syscall
	0x00400030	0x00000000	nop	23: nop #extra
	0x00400034	0x0000000d	break 0x00000000	24: break 0 #System.exit(0)

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10040000	l l e H	o W o	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0
0x10040020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0
0x10040040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0
0x10040060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0
0x10040080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0
0x100400a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0
0x100400c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0

0x10040000 (heap) Hexadecimal Addresses

Heap

Mars Messages Run I/O

```
Hello World
```

MIPS Assembly

MIPS Lab 1A

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400024	0x24020004	addiu $2,$0,0x00000004	19: li $v0, 4 #print code=
	0x00400028	0x3c011001	lui $1,0x00001001	20: la $a0, hello
	0x0040002c	0x34240000	ori $4,$1,0x00000000	
	0x00400030	0x0000000c	syscall	21: syscall
	0x00400034	0x3c011001	lui $1,0x00001001	23: lw $t3, test #test "ABCD"
	0x00400038	0x8c2b000d	lw $11,0x0000000d($1)	
	0x0040003c	0x214a0004	addi $10,$10,0x0000...	24: add $t2, $t2, 4 #heap ptr
	0x00400040	0x01402020	add $4,$10,$0	25: add $a0, $t2, $zero #$t2->$a0
	0x00400044	0xac8b0000	sw $11,0x00000000($4)	26: sw $t3, ($a0) #heap

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Val
0x10010000	l l e H	o W o	\n d l r	C B A \0	\0 \0 \0 D	
0x10010020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	
0x10010040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	
0x10010060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	
0x100100c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	

```
Error in /Users/jhdphd/Desktop/mips-Lab1B.asm line 23: Runtime exception at 0x00400038: fetch address not aligned on word boundary
```

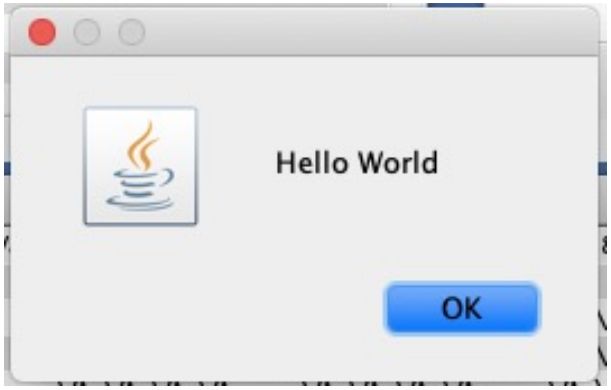
MIPS Assembly: GUI Out

GUI Output: Syscall **55**

MIPS Lab 1A

Use: Syscall **55**

MessageDialog	55	$a0 = address of null-terminated string that is the message to user $a1 = the type of message to be displayed: 0: error message, indicated by Error icon 1: information message, indicated by Information icon 2: warning message, indicated by Warning icon 3: question message, indicated by Question icon other: plain message (no icon displayed)



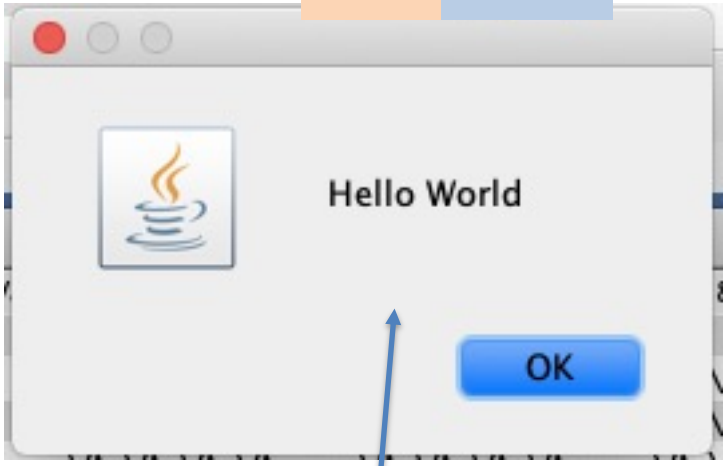

```
#output GUI msg
li $v0, 55 #GUI msg code
la $a0, hello
li $a1,1 #msg type is info
syscall
```

Memory Buffers: Output

MIPS Lab 1A

55

```
#output GUI msg
li $v0, 55 #GUI msg code
la $a0, hello
li $a1,1 #msg type is info
syscall
```



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	* * = =	\0 \n = =	l l e H	o W o	\n d l r
0x10010020	u p n I	t s t	g n i r	\0 \0 \0 :	\0 \0 \0 \0
0x10010040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100e0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x10010000 (.data) Hexadecimal Addresses Hexa

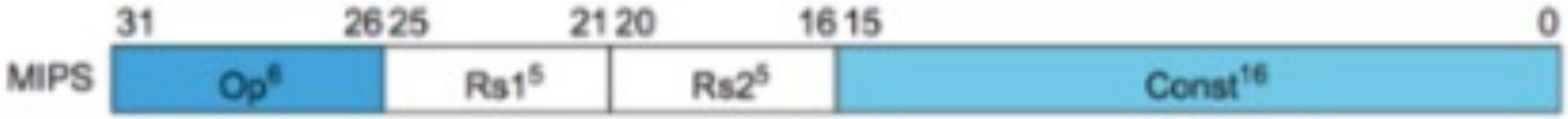
MIPS Assembly

22. load word: **lw** instruction

```
                +--------+-------+-------+-----------------------+
I-type format:  | 100011 | R_s   |  R_t  |        offset         |
                +--------+-------+-------+-----------------------+
Effects of the instruction: R_t <-- M{ [R_s] + [I_15]^16 || [I_15..0] }
                            PC <-- [PC] + 4
      (If an illegal memory address then exception processing)
Assembly format: lw R_t,offset(R_s)
```

27. *load upper immediate:* **lui** instruction

```
                +--------+-------+-------+-----------------------+
I-type format:  | 001111 | 00000 |  R_t  |        immediate      |
                +--------+-------+-------+-----------------------+
Effects of the instruction: R_t <-- [I_15-0] || 0^16; PC <-- [PC] + 4
Assembly format: lui R_t,immediate
```



MIPS Assembly

37. *no operation:* **nop** instruction

```
                +--------+-------+-------+-------+-------+--------+
R-type format   | 000000 | 00000 | 00000 | 00000 | 00000 | 000000 |
                +--------+-------+-------+-------+-------+--------+
```

Effects of the instruction: PC <-- [PC] + 4
Assembly format: **nop** (= **sll** R_0,**0** shift logical left 0)

Exception Handling

When a condition for any exception (overflow, illegal op-code, division by zero, etc.) occurs the following <u>hardware</u> exception processing is performed:

```
EPC <-- [PC]
                 /
                 |  0^28 || 1010    if illegal op-code (10)
Cause_Reg <--    |  0^28 || 1100    if overflow (12)
                 |  0^29 || 100     if illegal memory address (4)
                 \   ............   etc.
PC <-- 80000180_hex
```

MIPS Assembly

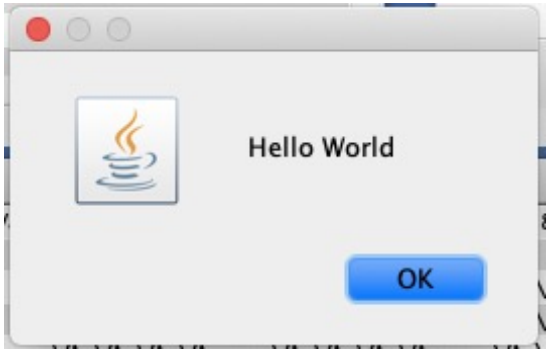
MIPS Lab 1A

Console prints

```
==*==
Hello World
Jeff D
Hello World
Jeff D

-- program is finished running --
```

Lab 1B



Lab 1A → Lab 1B

- Add 2 **LOOPS**
  - **Hello**
  - **Name**
- Add a **Header**
- Add Your Name
- Print **Header**

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Lab

MIPS

LAB 1 B

Hello World

- Add a Loop

- Add 2 **LOOPS**
  - **Hello**
  - **Name**
- Add a **Header**
- Add Your Name
- Print **Header**

Lab 1B: Add a Loop

MIPS Lab 1B

Hennessy & Patterson

PARTICIPATION ACTIVITY 2.7.3: Compiling a C while loop.

Start 2x speed

while

```
while (x == y) {
   // Loop body
}
```

```
Loop:  bne $s0, $s1, Exit
   # Loop body
   j Loop
Exit:
```

Loop: Hello

Java *For* Loop

```
//copy to heap (all words) using LOOP
for(int i=0; i<hello.length(); i++)
    heap[i] = charHello[i];
```

- By char

Hello loop

- By word

Assembly

loop:

bgtz

```
#loop for Hello
loop: sw $t1, ($t3) #store in memory: heap
addi $t2,$t2, 4 #adjust both ptrs
addi $t3,$t3, 4
lw $t1, ($t2) #next word
subi $t0,$t0, 1 #decr count
bgtz $t0,loop # !N and !Z
#end Loop
```

Lab 1B: If-Then-Else

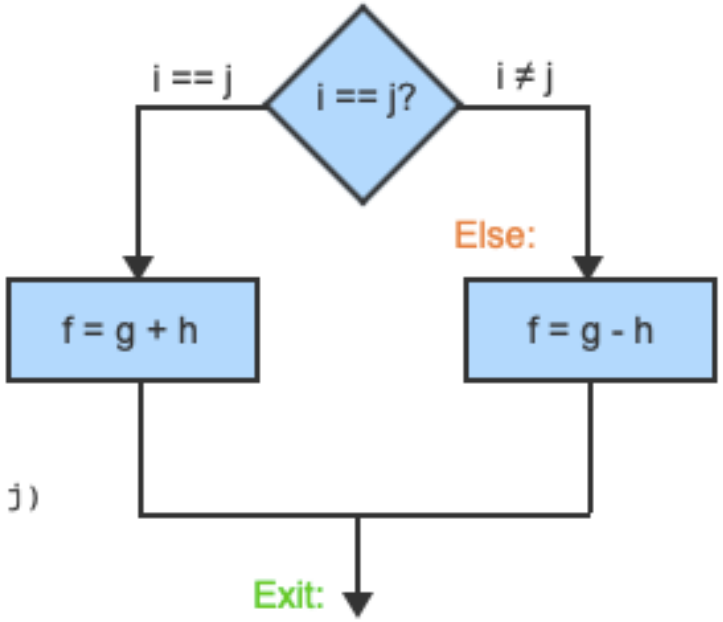
MIPS Lab 1B

PARTICIPATION ACTIVITY 2.7.1: Example of compiling if-then-else into cond Hennessy & Patterson D Figure 2.9).

Start 2x speed

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

```
bne $s3, $s4, Else    # go to Else if i ≠ j
add $s0, $s1, $s2     # f = g + h  (skipped if i ≠ j)
j  Exit               # go to Exit
Else:  sub $s0, $s1, $s2   # f = g - h  (skipped if i == j)
Exit:
```



CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

MIPS Assembly

MIPS Lab 1B

```
## Lab 1B -- Hello World
## by Jeff Drobman
##version date: 2-4-20
##add:  header, loop, macro, GUI out
#register map:
#$t1=string data, $t2=string ptr, $t3=heap ptr
#$t0=loop count (3)
```

```
##version: 1B- Loop >10-10-19
.data
header: .asciiz  "==**==\n"
.align 2 #align to word
hello: .asciiz  "Hello World\n"
.align 2 #align to word
name:  .asciiz  "Jeff\n"
#define
.eqv heap, 0x10040000
.text
#setup Loop: ctr=$t0
li $t0, 3 #N=3 (not 2!)
#init
lw $t1, hello #data
la $t2, hello #ptr
la $t3, heap #0x10040000
#loop for Hello
loop: sw $t1, ($t3) #store in memory: heap
addi $t2,$t2, 4 #adjust ptrs
addi $t3,$t3, 4
lw $t1, ($t2) #next word
subi $t0,$t0, 1 #count--
bgtz $t0,loop
```

loop

```
#end Loop
#name -> heap
lw $t1, name #data
la $t2, name #ptr
sw $t1, ($t3) #heap
#--end write to heap
#--print on console using "Syscall"
li $v0, 4 #print str code
la $a0, header
syscall
la $a0, hello
syscall
la $a0, name
syscall
la $a0, heap #print entire string in heap
syscall
nop
li $v0, 10 #stop code
syscall #stop
#break 0 #System.exit(0)
```

Add name

?

done

```
.macro done
li $v0, 10 #stop code
syscall #stop
.end_macro
```

MIPS Assembly

MIPS Lab 1B

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10040000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x10040000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x6666654a
$t2	10	0x10010018
$t3	11	0x1004000c

```
beq $t1,$t2,label        Branch if equal : Branch to statement at label's address if $t1 and $t2 are equal
bgez $t1,label           Branch if greater than or equal to zero : Branch to statement at label's address
bgezal $t1,label         Branch if greater then or equal to zero and link : If $t1 is greater than or equa
bgtz $t1,label           Branch if greater than zero : Branch to statement at label's address if $t1 is gr
blez $t1,label           Branch if less than or equal to zero : Branch to statement at label's address if
bltz $t1,label           Branch if less than zero : Branch to statement at label's address if $t1 is less
bltzal $t1,label         Branch if less than zero and link : If $t1 is less than or equal to zero, then se
bne $t1,$t2,label        Branch if not equal : Branch to statement at label's address if $t1 and $t2 are n
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

MIPS Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER

MIPS Lab 1B

Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24080003	addiu $8,$0,0x00000003	14: li $t0, 3 #N=3 (not 2!)
	0x00400004	0x3c011001	lui $1,0x00001001	16: lw $t1, hello #data
	0x00400008	0x8c290008	lw $9,0x00000008($1)	
	0x0040000c	0x3c011001	lui $1,0x00001001	17: la $t2, hello #ptr
	0x00400010	0x342a0008	ori $10,$1,0x00000008	
	0x00400014	0x3c011004	lui $1,0x00001004	18: la $t3, 0x10040000 #0x10040000
	0x00400018	0x342b0000	ori $11,$1,0x00000000	
	0x0040001c	0xad690000	sw $9,0x00000000($11)	20: loop: sw $t1, ($t3) #store in memory: heap
	0x00400020	0x214a0004	addi $10,$10,0x0000...	21: addi $t2,$t2, 4 #adjust ptrs

Labels

Label	Address ▲
mips-Lab1B.asm	
loop	0x0040001c
header	0x10010000
hello	0x10010008
name	0x10010018

☑ Data ☑ Text

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10040000	l l e H	o W o	\n d l r	f f e J	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x10040000 (heap) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☑ ASCII

Mars Messages | Run I/O

```
==*==
Hello World
Jeff
Hello World
Jeff
-- program is finished running --
```

Clear

Loop: Name

Java *While* Loop

❖Simulating a *Heap*

```
public class Lab1Hello122 {
//static global DATA
   static String hello = "Hello World!";
   static String name = "Jeff Drobman0"; //asciiz
   static String prompt = "Input name";
//main method
   public static void main(String[] args) {
      //simulate a "heap" = dynamic DATA segment
      int size = 1000;
      char[] heap = new char[size]; //reserve 1000 bytes
      char[] charHello = hello.toCharArray();
      char[] charName = name.toCharArray();
      int ix = 0; //heap index
```

Loop: Name

Java | *While* Loop | ***Indefinite***

```
//indefintite Loop for var-length name (until 0 char)
int ix2 = 0; //new index
String sx = ""; //use a String
while(true) {
    char chx = charName[ix2++];
    if(chx == '0') break; //check for null 0
    System.out.print(chx); //log char
    heap[ix++] = chx; //store in heap
    sx += chx; //append to heap string
}//end while
```

Name loop

Assembly

loop:

bgtz

- By char | Use **LB**, **SB**

✓ Check for null **0**

Loop: Name

Java *While* Loop

```
//OUTPUT
System.out.println(hello);//console
JOptionPane.showMessageDialog(null, hello);//GUI box
System.out.println("Heap= ");
for(int i=0;i<ix;i++) System.out.print(heap[i]);
System.out.println("<end heap>"); //clean line
```

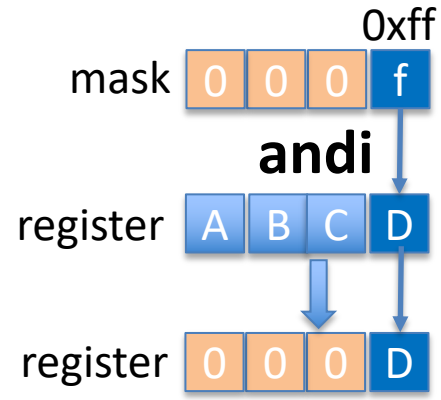
```
 ----jGRASP exec: java Lab1Hello122
Jeff Drobman@
Hello World!
Heap=
Hello World!>Jeff Drobman<end heap>
Howdy jd
```

Case 1: Mask

While Loop | Extract a byte | MIPS Lab 1B

mask

```
8  .eqv heapHi, 0x1004
9  .eqv mask0, 0x000000ff #byte0
```



Use mask

$t4

```
38 #name -> heap
39 lw $t1, name #data
40 la $t2, name #ptr
41 Loop_name:
42 sw $t1, ($t3) #heap
43 addi $t2,$t2, 4 #adjust both ptrs
44 addi $t3,$t3, 4
45 lw $t1, ($t2) #next word
46 andi $t4,$t1,mask0 #mask byte0
47 bnez $t4,Loop_name #test byte0=0
48 #repeat block for bytes 1, 2, 3
49 #--end write to heap
```

Test byte=0

Fall out of loop

compare

```
46 bnez $t1,Loop_name #test whole word=0000
47 #--end write to heap
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

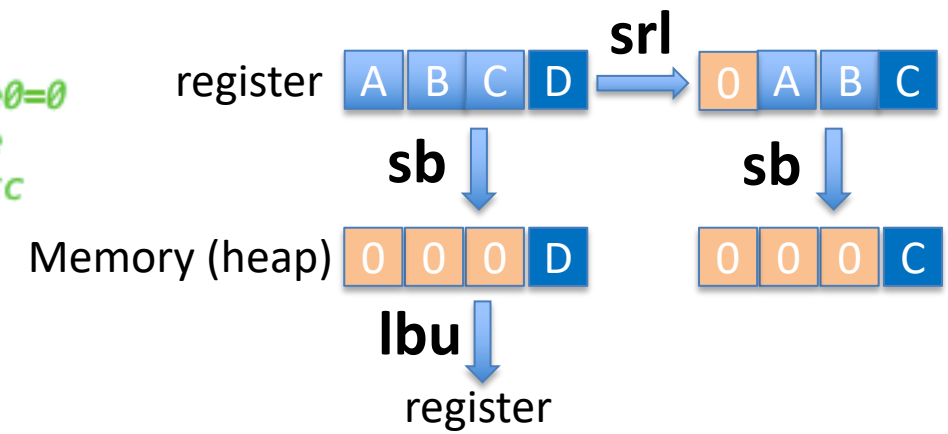
Case 2: Memory (Lb/Sb)

While Loop | Extract a byte | MIPS | Lab 1B

mask

```
8   .eqv heapHi, 0x1004
9   .eqv mask0, 0x000000ff #byte0
```

```
49  #try load from memory 1 byte at a time
50  la $t5,24($t3) #leave a gap in heap of 24 bytes
51  sb $t1,($t5)
52  lbu $t4,($t5)
53  beqz $t4,exit #test byte0=0
54  srl $t6,$t1,8 #next byte
55  #repeat for 2nd byte, etc
56  sb $t6,1($t5)
57  lbu $t4,1($t5)
58  bnez $t4,Loop_name
59  exit:
60  #--end write to heap
```




```
sb $t1,-100($t2)        Store byte : Store the low-order 8 bits of $t1 into the effective memory byte add
```

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)
0x10040000	l l e H	o W o	\n d l r	f f e J	o r D	n a m b
0x10040020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 D	\0 \0 m b	\0 \0 \0 \n	\0 \0 \0 \0
0x10040040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10040080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100400e0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

Case Switch (Java)


```java
//CASE
int cNum = 1;
switch(cNum){
    case 1: int x = 1;
        break;
    case 2: x = 2;
        break;
}
```

What's this? A ***Branch Table*** Case Switch

```
#checks input
check: nop
beq $a1, -2, cancel
beq $a1, -3, nodata
beq $a1, -4, exceed
jr $ra
```

```
.eqv cNum, 2 #case 1
```

```
#check: Case= 1 or 2
li $s0,cNum #$s0 <-cNum
beq $s0,2,case2
#else
case1:
```

Case 2: Memory (Lb/Sb)


```
case2:
la $t5,24($t3) #leave a gap in heap of 24 bytes
sb $t1,($t5)
lbu $t4,($t5)
beqz $t4,exit #test byte0=0
srl $t6,$t1,8 #next byte
#repeat for 2nd byte, etc
sb $t6,1($t5)
lbu $t4,1($t5)
bnez $t4,Loop_name
exit:
#--end write to heap
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

MIPS Code Puzzle

DR JEFF SOFTWARE
INDIE APP DEVELOPER

OnlineGDB beta
online compiler and debugger for c/c++

code. compile. run. debug. share.

IDE
My Projects
Classroom new
Learn Programming
Programming Questions
Sign Up
Login

SPONSOR Sendbird — Sendbird Calls Voice API & Video API: Increase in-app engagement with voice and video

Fork this | Run | Lan

main.S

```
sw $t4, 0($t1)
li $t4, 0b00000000 # turn off all leds
sw $t4, 0($t1)

switch_3:
li $t4, 0b00000001 # turn on first led
# *t1 = t4
sw $t4, 0($t1)
shift_t4: #loop to make cycle through all 8 LEDs
sll $t4, $t4, 1
sw $t4, 0($t1)
beq $t4, $0, end # put END condition FIRST to avoid looping infinitely
jump_shift_t4:
j shift_t4

end:
j end
nop
```

While(true)

SLEEP/WAIT

MIPS Assembly

31. *branch on greater than zero:* **bgtz** instruction

```
                +--------+-------+-------+-----------------------+
I-type format:  |000111  |  Rs   |  00000|        offset         |
                +--------+-------+-------+-----------------------+
```

Effects of the instruction:

if $[R_s] > 0$ then $PC \leftarrow [PC] + 4 + ([I_{15}]^{14} \,||\, [I_{15..0}] \,||\, 0^2)$

else $PC \leftarrow [PC] + 4$

Assembly format: **bgtz R_s,offset**

34. *jump and link:* **jal** instruction

```
               +--------+-------------------------------------+
J-type format  |000011  |              jump_target            |
               +--------+-------------------------------------+
```

Effects of the instruction: $R_{31} \leftarrow [PC] + 4$

$PC \leftarrow [PC_{31..28}] \,||\, [I_{25..0}] \,||\, 0^2$

Assembly format: **jal jump_target**

36. *jump and link register:* **jalr** instruction

```
               +--------+-------+-------+-------+-------+--------+
R-type format  | 000000 |  Rs   | 00000 | Rd    | 00000 | 001001 |
               +--------+-------+-------+-------+-------+--------+
```

Effects of the instruction: $R_d \leftarrow [PC] + 4$; $PC \leftarrow [R_s]$

Assembly format: **jalr R_d,R_s**

MIPS Assembly

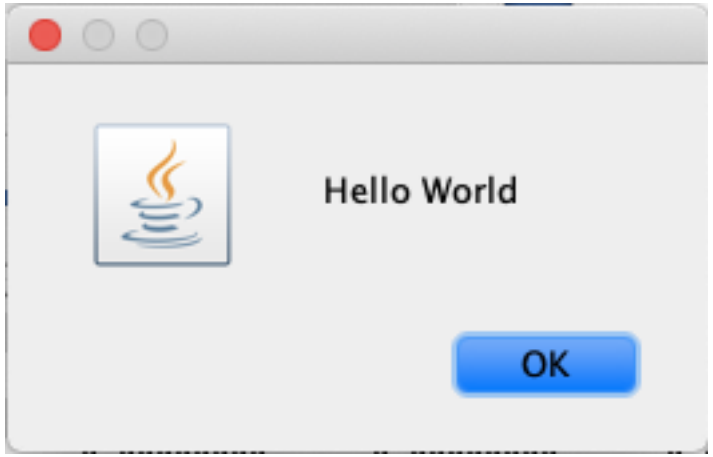
MIPS Lab 1B

Console prints

```
==*==
Hello World
Jeff D
Hello World
Jeff D

-- program is finished running --
```

GUI



Lab 1B → Lab 1C

- SUB
- GUI IN

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Lab

DR JEFF SOFTWARE
INDIE APP DEVELOPER

MIPS

LAB 1 C

Hello World

- Sub
- GUI In

Subroutines

Java

write_heap($a0,$a1, $a2,$a3);

Assembly

loop:

bgtz

```
done #macro for exit
#end of main
#subroutine: write String to heap($a0..$a3)
write_heap: nop #sub label
loop: sw $a1, ($a3) #count N= $a0
addi $a2,$a2, 4 #adjust ptrs
addi $a3,$a3, 4
lw $a1, ($a2) #load next word
subi $a0,$a0, 1 #count--
bgtz $a0,loop
#return
jr $ra
```

MIPS Assembly


```
## Lab 1C -- Hello World
## by Jeff Drobman
##version: 1C- Loop >9-22-20
##add:  sub, GUI IN: prompt+INbuf
#register map:
#$a1=string data, $a2=string ptr, $a3=heap ptr
#$a0=loop count (3)
.data
```

```
.data
header: .asciiz  "==**==\n"
.align 2 #align to word
hello: .asciiz  "Hello World\n"
.align 2 #align to word
name:  .asciiz  "Jeff D\n"
prompt: .asciiz  "Input string:"
#define
.eqv heap, 0x10040000
.eqv in_buf, 0x10040020 #input buffer
#macros
.macro done
li $v0, 10 #stop code
syscall #stop
.end_macro
```

MIPS Assembly

➢ Add this

MIPS Lab 1C

Setup args

```
#code
.text
#args(4) $a0-$a3 for call "write_heap"
li $a0, 3 #N=3
lw $a1, hello #data
la $a2, hello #ptr
la $a3, heap #0x10040000
#call sub for hello
jal write_heap
#name -> heap
li $a0, 2 #N=2
lw $a1, name #data
la $a2, name #ptr
#$a3->heap has been updated
#call sub for name
jal write_heap
#--end write to heap
```

CALL

CALL

$ra <- PC

args
$a0..$a3

Sub

```
#end of main
#subroutine: write String to heap($a0..$a3)
write_heap: nop #sub label
loop: sw $a1, ($a3) #count N= $a0
addi $a2,$a2, 4 #adjust ptrs
addi $a3,$a3, 4
lw $a1, ($a2) #load next word
subi $a0,$a0, 1 #count--
bgtz $a0,loop
#return
jr $ra
```

I/O

Output

Java

```
JOptionPane.showMessageDialog(null, msg);
```

Assembly

```
#output GUI msg
li $v0, 55 #GUI msg code
la $a0, hello
li $a1,1 #msg type is info
syscall
```

SPIM code

55

Input

```
//INPUT
String name = JOptionPane.showInputDialog(prompt);//GUI box
```

Assembly

```
#INput GUI msg
li $v0, 54 #GUI msg code
la $a0, prompt
la $a1, in_buf
li $a2,20 #max input length
syscall
```

54

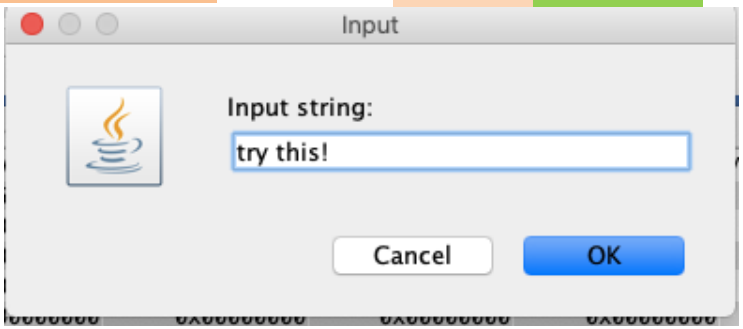
MIPS Assembly: GUI In

GUI Input: Syscall **54**

MIPS Lab 1C

➢ Add this

Use: Syscall **54**



InputDialogString	54	$a0 = address of null-terminated string that is the message to user $a1 = address of input buffer $a2 = maximum number of characters to read	*See Service 8 note below table* $a1 contains status value 0: OK status. Buffer contains the input string. -2: Cancel was chosen. No change to buffer. -3: OK was chosen but no data had been input into field. No change to buffer. -4: length of the input string exceeded the specified maximum. Buffer contains the maximum allowable input string plus a terminating null.

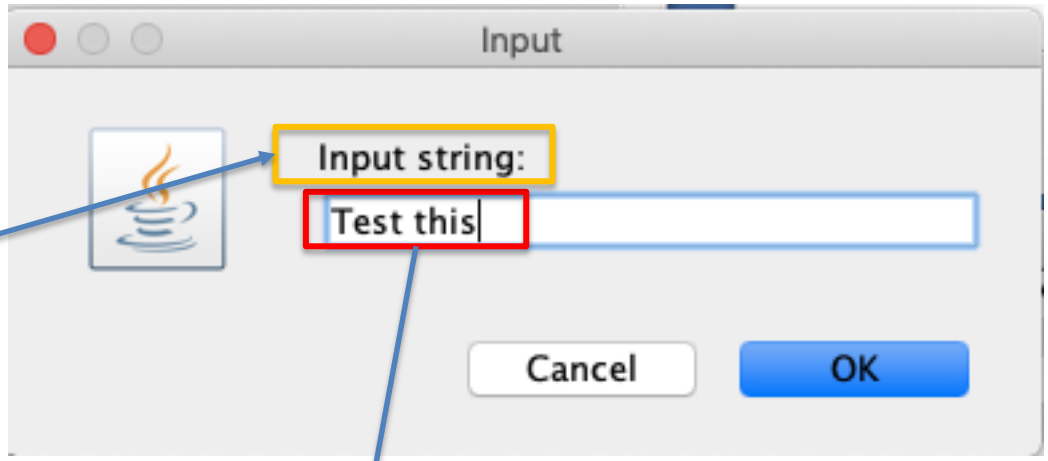
```
#INput GUI msg
li $v0, 54 #GUI msg code
la $a0, prompt
la $a1, in_buf
li $a2,20 #max input length
syscall
```

Memory Buffers: Input

GUI Input: Syscall 54

MIPS Lab 1C

```
.data
header: .asciiz  "==**==\n"
.align 2 #align to word
hello: .asciiz  "Hello World\n"
.align 2 #align to word
name:  .asciiz  "Jeff D\n"
prompt: .asciiz  "Input string:"
#define
.eqv heap, 0x10040000
.eqv in_buf, 0x10040020 #input buffer
```




```
.eqv in_buf, 0x10040020 #input buffer
```

INPUT BUFFER

54

```
#INput GUI msg
li $v0, 54 #GUI msg code
la $a0, prompt
la $a1, in_buf
li $a2,20 #max input length
syscall
```

Address	Value (+0)	Value (+4)	Value (+8)	V
0x10040000	l l e H	o W o	\n d l r	
0x10040020	t s e T	i h t	\0 \0 \n s	
0x10040040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	
0x10040060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	
0x10040080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	
0x100400a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	
0x100400c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	
0x100400e0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	

0x10040000 (heap)

Check for Error

What's this? A ***Branch Table***

```
#checks input
check: nop
beq $al, -2, cancel
beq $al, -3, nodata
beq $al, -4, exceed
jr $ra
```

Echo Name on Console


```
//INPUT
String name = JOptionPane.showInputDialog(prompt);//GUI box
//echo
System.out.println("Hello" + name);//console
```

```
----jGRASP exec: java Lab1Hello122
Hello World!
Hello Jeff
```

Lab

LAB 1 D

Hello World

Port to

ARM

Hello World in Assembly

essors blog › "Hello World" in Assembly

ARM Gnu

```
    .syntax unified

    @ -------------------------------
.global main
main:
    @ Stack the return address (lr) in addition to a dummy register (ip) to
    @ keep the stack 8-byte aligned.
    push    {ip, lr}

    @ Load the argument and perform the call. This is like 'printf("...")' in C.
    ldr     r0, =message
    bl      printf

    @ Exit from 'main'. This is like 'return 0' in C.
    mov     r0, #0    @ Return 0.

    @ Pop the dummy ip to reverse our alignment fix, and pop the original lr
    @ value directly into pc — the Program Counter — to return.
    pop     {ip, pc}

    @ -------------------------------
    @ Data for the printf calls. The GNU assembler's ".asciz" directive
    @ automatically adds a NULL character termination.
message:
    .asciz "Hello, world.\n"
```

Call printf

Hello World in Assembly

cessors blog › "Hello World" in Assembly

ARM Gnu

+ New

We can assemble and run this program using the following (on an Arm Linux-like platform):

```
gcc -o hello_world hello_world.s
$ ./hello_world
```

You should then see the text "Hello, world." on the console.

If you're using a cross-compiler (such as RVCT or the Code Sourcery edition of GCC) you'll need to run the first step on your PC — probably substituting `gcc` with something like `arm-none-linux-gnueabi-gcc` — and then copy the output binary to an Arm target before running the program itself.

Gnu Assembly: Hello World


```
        .data
str:    .asciz  "Hello World\n" @ Define a null-terminated string

        .text
        .globl  main
        /* This is the beginning of the main() function.
           It will print "Hello World" and then return.
        */
main:   stmfd   sp!,{lr}        @ push return address onto stack
        ldr     r0, =str        @ load pointer to format string
        bl      printf          @ printf("Hello World\n");
        mov     r0, #0          @ move return code into r0
        ldmfd   sp!,{lr}        @ pop return address from stack
        mov     pc, lr          @ return from main
```

LISTING 2.1 "Hello World" program in ARM assembly

```
#include <stdio.h>
static char str[] = "Hello World\n";
int main()
{
  printf(str);
  return 0;
}
```

LISTING 2.2 "Hello World" program in C.

Hello World-Assembly *Listing*

ARM Gnu | Lab 1

```
ARM GAS  hello.S                    page 1

   Addr Code            Source
 1                      .data
 2 0000 48656C6C  str:  .asciz  "Hello World\n"
 2      6F20576F
 2      726C640A
 2      00
 3
 4                      .text
 5                      .globl  main
 6                      /* This is the beginning of the main() function.
 7                         It will print Hello World'' and then return.
 8                      */
 9 0000 00402DE9  main: stmfd   sp!,{lr} @ push return address onto stack
10 0004 0C009FE5        ldr     r0, =str @ load pointer to format string
11 0008 FEFFFFEB        bl      printf   @ printf("Hello World - %d\n",i);
12 000c 0000A0E3        mov     r0, #0   @ move return code into r0
13 0010 0040BDE8        ldmfd   sp!,{lr} @ pop return address from stack
14 0014 0EF0A0E1        mov     pc, lr   @ return from main
14      00000000

ARM GAS  hello.S                    page 2


DEFINED SYMBOLS
            hello.S:2      .data:0000000000000000 str
            hello.S:9      .text:0000000000000000 main
            hello.S:9      .text:0000000000000000 $a
            hello.S:14     .text:0000000000000018 $d

UNDEFINED SYMBOLS
printf
```

LISTING 2.3 "Hello World" assembly listing.

Assembler *Directives*


```
        .data
i:      .word   0
j:      .word   1
fmt:    .asciz  "Hello\n"
ch:     .byte   'A','B',0
ary:    .word   0,1,2,3,4
```

(A) Declarations in assembly

```
static int i = 0;
static int j = 1;
static char fmt[] = "Hello\n";
static char ch[] = {'A','B',0};
static int ary[] = {0,1,2,3,4};
```

(B) Declarations in C

FIGURE 2.1 Equivalent static variable declarations in assembly and C.

Lab 1D – ARMsim


```
R0        :000010a0
R1        :00000001
R2        :65707954
R3        :00004014
R4        :00000037
R5        :00000002
R6        :00000000
R7        :00000000
R8        :00000000
R9        :00000000
R10(sl):00000000
R11(fp):00000000
R12(ip):00000000
R13(sp):00011400
R14(lr):00001024
R15(pc):00001058
------------------
CPSR Register
Negative(N):0
Zero(Z)     :1
Carry(C)    :1
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
Thumb(T)    :0
CPU Mode    :System
------------------
0x600000df
```

Lab 1D – ARMsim


```
                        .data
00001098:2A2A3D3D       header: .asciz "==**==\n"
        :000A3D3D
                        .align 2 @align to word
000010A0:6C6C6548       hello: .asciz "Hello World\n"
        :6F57206F
        :0A646C72
        :00
```

```
00001000
Word Size: 8Bit 16Bit 32Bit

00001000  03 00 A0 E3 80 10 9F E5 00 20 91 E5 01 39 A0 E3 11 00  .. ã...å. .å.9 ã..
00001012  00 EB 02 00 A0 E3 70 10 9F E5 00 20 91 E5 0D 00 00 EB  .ë.. ãp..å. .å ..ë
00001024  68 00 9F E5 02 00 00 EF 58 00 9F E5 02 00 00 EF 54 00  h..å...ïX..å...ïT.
00001036  9F E5 02 00 00 EF 01 09 A0 E3 02 00 00 EF 37 40 A0 E3  .å...ï.  ã...ï7@ ã
00001048  3C 00 9F E5 01 10 A0 E3 37 00 00 EF 00 F0 20 E3 11 00  ...å.. ã7..ï.ð ã..
0000105A  00 EF 00 50 A0 E3 00 20 83 E5 04 10 81 E2 04 30 83 E2  .ï.P ã. .å...â.0.â
0000106C  00 20 91 E5 01 50 85 E2 01 00 40 E2 00 00 50 E3 F7 FF  . .å.P â..@â..Pã.ÿ
0000107E  FF CA 1E FF 2F E1 00 F0 20 E3 1E FF 2F E1 A0 10 00 00  ÿÊ.ÿ/á.ð ã.ÿ/á ...
00001090  B0 10 00 00 98 10 00 00 3D 3D 2A 2A 3D 3D 0A 00 48 65  ..........**.. .He
000010A2  6C 6C 6F 20 57 6F 72 6C 64 0A 00 00 00 00 47 61 62 65  llo World ....Gabe
000010B4  20 4D 0A 00 54 79 70 65 20 49 6E 70 75 74 3A 00 81 81   M .Type Input:...
000010C6  81 81 81 81 81 81 81 81 81 81 81 81 81 81 81 81 81 81  ..................
```

```
00004000
Word Size: 8Bit 16Bit 32Bit

00004000  48 65 6C 6C 6F 20 57 6F 72 6C 64 0A 47 61 62 65 20 4D  Hello World Gabe M
00004012  0A 00 81 81 81 81 81 81 81 81 81 81 81 81 81 81 81 81  ..................
```

Lab 1D – ARMsim

Lab 1D (ARM)

perfect/very good, and

1. Loop-- good use of "cmp" to set up branch.
2. GUI output: what happens when you execute SWI 55? You used reg "v1" but there is no such ARM reg.

```
@output GUI msg
mov v1, #55 @GUI msg code
ldr a1, =hello
mov a2, #1 @msg type is info
svc 55 @syscall
```

Svc 55

```
@subrountine:write String to heap($a0..$a3)
write_heap:
mov r5, #0 @counter
Loop: str a3, [a4] @count N= a1
add a2, a2, #4 @adjust ptrs
add a4, a4, #4
ldr a3, [a2] @load next word
add r5, #1
sub a1, #1 @count--
cmp a1, #0
Bgt Loop @EQ/NE/GE?
BX LR @return
@-----
printf: nop @stub
BX LR @return
.end
```

Cmp a1, #0

Lab

LAB 6 MIPS

Fibonacci Sequence

Lab 6: Fibs in Arrays

Hennessy & Patterson

PARTICIPATION ACTIVITY 2.3.5: Example of compiling an assignment when an operand is in memory.

Start 2x speed

g = h + A[8];

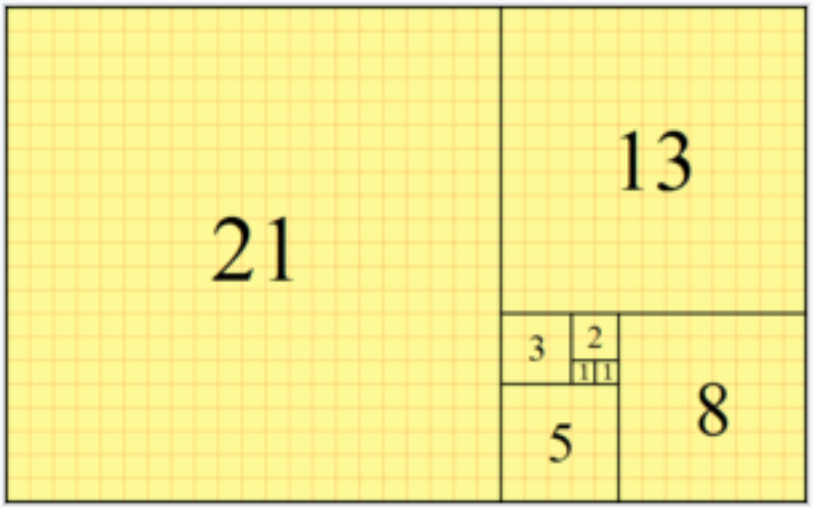

```
# Temporary reg $t0 gets A[8]
lw    $t0, 8($s3)
```

8 + 5000

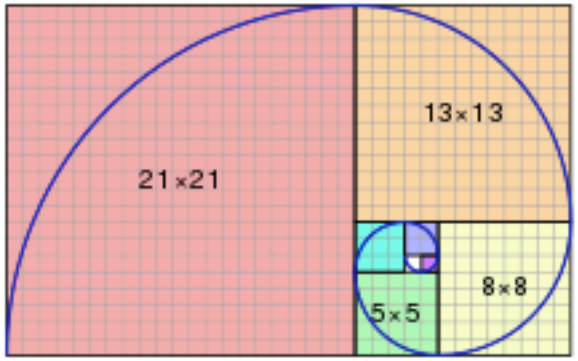
```
# g = h + A[8]
add   $s1, $s2, $t0
```

35 + 25

Fibonacci Sequence



A tiling with squares whose side lengths are successive Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13 and 21.



The Fibonacci spiral: an approximation of the golden spiral created by drawing circular arcs connecting the opposite corners of squares in the Fibonacci tiling; (see preceding image)

The first 21 Fibonacci numbers F_n are:[2]

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

Fibonacci in Java

Lab 2

```
public class Lab2Fibs122 {
   public static void main(String[] args) {
      int N = 12;
      int[] F = new int[N];
      F[0]=1; F[1]=1;
      System.out.print("Fibs: " +F[0] +" " +F[1] +" ");
      for(int i=2; i<N; i++) {
         F[i] = F[i-1] + F[i-2];
         System.out.print(F[i] +" ");
         }//end for
      System.out.println("\n-----");
      printit(F);//call printit
   }//end main
   //printit sub
   static void printit(int[] arr) {
      System.out.print("Fibs: ");
      for(int x: arr)
         System.out.print(x +" ");
      System.out.println();
   }//end printit
}//end class
```

Print inline

Print sub

```
----jGRASP exec: java Lab2Fibs122
Fibs: 1 1 2 3 5 8 13 21 34 55 89 144
-----
Fibs: 1 1 2 3 5 8 13 21 34 55 89 144
```

Fibonacci

MIPS

Compute first twelve ***Fibonacci*** numbers and put in array, then print

```
        .data
fibs:   .word   0 : 12      # "array" of 12 words to contain fib values
size:   .word   12          # size of "array"
        .text
        la    $t0, fibs     # load address of array
        la    $t5, size     # load address of size variable
        lw    $t5, 0($t5)   # load array size
        li    $t2, 1        # 1 is first and second Fib. number
        add.d $f0, $f2, $f4 ?
        sw    $t2, 0($t0)   # F[0] = 1
        sw    $t2, 4($t0)   # F[1] = F[0] = 1
        addi  $t1, $t5, -2  # Counter for loop, will execute (size-2) times
loop:   lw    $t3, 0($t0)   # Get value from array F[n]
        lw    $t4, 4($t0)   # Get value from array F[n+1]
        add   $t2, $t3, $t4 # $t2 = F[n] + F[n+1]
        sw    $t2, 8($t0)   # Store F[n+2] = F[n] + F[n+1] in array
        addi  $t0, $t0, 4   # increment address of Fib. number source
        addi  $t1, $t1, -1  # decrement loop counter
        bgtz  $t1, loop     # repeat if not finished yet.
        la    $a0, fibs     # first argument for print (array)
        add   $a1, $zero, $t5 # second argument for print (size)
        jal   print         # call print routine.
        li    $v0, 10       # system call for exit
        syscall             # we are out of here.
```

Loop

1
1
2
3
5
8
13
21
34
55
89
144

Fibonacci

MIPS

Compute first twelve ***Fibonacci*** numbers and put in array, then print

```
#########  routine to print the numbers on one line.

        .data
space:.asciiz   " "              # space to insert between numbers
head: .asciiz   "The Fibonacci numbers are:\n"
        .text
print:add  $t0, $zero, $a0   # starting address of array
      add  $t1, $zero, $a1   # initialize loop counter to array size
      la   $a0, head         # load address of print heading
      li   $v0, 4            # specify Print String service
      syscall                # print heading
out:  lw   $a0, 0($t0)       # load fibonacci number for syscall
      li   $v0, 1            # specify Print Integer service
      syscall                # print fibonacci number
      la   $a0, space        # load address of spacer for syscall
      li   $v0, 4            # specify Print String service
      syscall                # output string
      addi $t0, $t0, 4       # increment address
      addi $t1, $t1, -1      # decrement loop counter
      bgtz $t1, out          # repeat if not finished
      jr   $ra               # return
```

Loop

1
1
2
3
5
8
13
21
34
55
89
144

ALU Ops

MARS


```
add $t1,$t2,$t3          Addition with overflow : set $t1 to ($t2 plus $t3)
add.d $f2,$f4,$f6        Floating point addition double precision : Set $f2 to double-precision floating p
add.s $f0,$f1,$f3        Floating point addition single precision : Set $f0 to single-precision floating p
addi $t1,$t2,-100        Addition immediate with overflow : set $t1 to ($t2 plus signed 16-bit immediate)
addiu $t1,$t2,-100       Addition immediate unsigned without overflow : set $t1 to ($t2 plus signed 16-bit
addu $t1,$t2,$t3         Addition unsigned without overflow : set $t1 to ($t2 plus $t3), no overflow
and $t1,$t2,$t3          Bitwise AND : Set $t1 to bitwise AND of $t2 and $t3
andi $t1,$t2,100         Bitwise AND immediate : Set $t1 to bitwise AND of $t2 and zero-extended 16-bit im
```

Fibonacci: Dr Jeff

MIPS


```
## Lab 2 -- Fibonacci
## by Jeff Drobman
##version: 1.0 >11-21-19
.data
header: .asciiz  "Fibonacci sequence by Jeff D\n"
.align 2
fibs: .word  0:12
.align 2
size:  .word  12
space:  .word  0x20 #space
#define
#.eqv heap, 0x10040000 -not used
#macros
.macro done
li $v0, 10 #stop code
syscall #stop
.end_macro
#code
.text
#set up pointers
la $t0, fibs #ptr
lw $t5, size #final
subu $t1,$t5,2 #counter
#init 1st 2 numbers (1, 1)
li $t2,1
li $t3,1
sd $t2,($t0) #both 1's
addi $t0,$t0,8 #incr ptr
loop_main:
add $t4,$t2,$t3 #next fib
```

```
Fibonacci sequence by Jeff D
1 1 2 3 5 8 13 21 34 55 89 144
-- program is finished running --
```

1
1
2
3
5
8
13
21
34
55
89
144

pseudo

```
sd $t1,($t2)
sd $t1,-100($t2)
sd $t1,100000
sd $t1,100000($t2)
sd $t1,label
sd $t1,label($t2)
sd $t1,label+100000
sd $t1,label+100000($t2)
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Fibonacci: Dr Jeff

MIPS

Loop

```
sw $t4,($t0)
addi $t0,$t0,4 #incr ptr
move $t2,$t3
move $t3,$t4
subi $t1,$t1,1 #counter
bgtz $t1,loop_main
#call sub for print
jal print
done #macro for exit
#---end of main---
#subroutine: print($t0=ptr, $t5=size)
print: nop #sub label
#print header
li $v0, 4
la $a0, header
syscall
#print fibs
la $t0, fibs #ptr
loop_prt:
li $v0, 1 #print int code
lw $a0, ($t0)
syscall
li $v0, 11 #print char code
lw $a0, space
syscall
addi $t0,$t0,4 #incr ptr
subi $t5,$t5, 1 #count--
bgtz $t5,loop_prt
#return
jr $ra
```

```
Fibonacci sequence by Jeff D
1 1 2 3 5 8 13 21 34 55 89 144
-- program is finished running --
```

1
1
2
3
5
8
13
21
34
55
89
144

Page 2/2

Fibonacci: Dr Jeff

MIPS

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	o b i F	c c a n	e s i	n e u q	b e c	e J y	D f f	\0 \0 \0 \n
0x10010020	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 \b	\0 \0 \0 \r	\0 \0 \0 .
0x10010040	\0 \0 \0 "	\0 \0 \0 7	\0 \0 \0 Y	\0 \0 \0 .	\0 \0 \0 \f	\0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

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

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	1868720454	1667457390	1702043753	1852142961	1646290275	1699356793	1142974054	10
0x10010020	1	1	2	3	5	8	13	21
0x10010040	34	55	89	144	12	32	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0

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

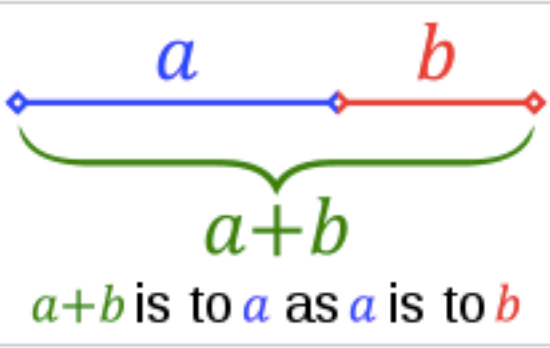
Fibonacci—Golden Ratio

Golden ratio

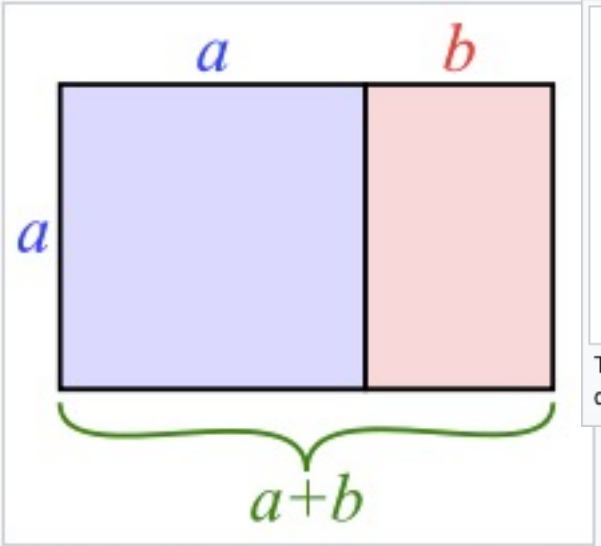
From Wikipedia, the free encyclopedia

$$\frac{a+b}{a} = \frac{a}{b} \stackrel{\text{def}}{=} \varphi,$$

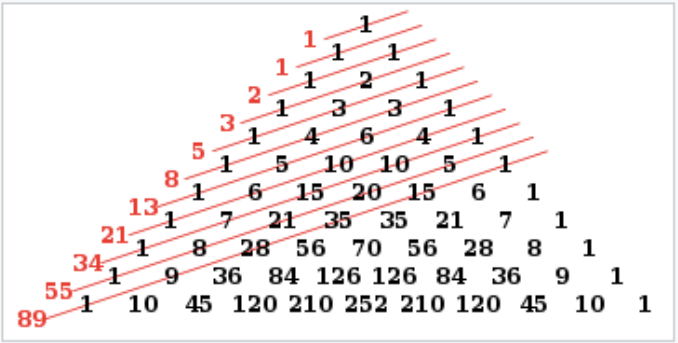
$$\varphi = \frac{1+\sqrt{5}}{2} = 1.6180339887\ldots$$



Line segments in the golden ratio



A golden rectangle with longer side ***a*** and shorter side ***b***, when placed adjacent to a square with sides of length ***a***, will produce a similar golden rectangle with longer side ***a + b*** and shorter side ***a***. This illustrates the relationship $\frac{a+b}{a} = \frac{a}{b} \equiv \varphi.$



The Fibonacci numbers are the sums of the "shallow" diagonals (shown in red) of Pascal's triangle.

Fibonacci in Java

1st 30

Lab 2

```
 ----jGRASP exec: java Fibs
Fibs as gen:
1  1  2  3  5  8  13  21  34  55  89  144  233
377  610  987  1597  2584  4181  6765  10946  17711  28657  46368  75025
121393  196418  317811  514229  832040
final 3: 317811  514229  832040
-----
Fibs from array: 30
1  1  2  3  5  8  13  21  34  55  89  144  233
377  610  987  1597  2584  4181  6765  10946  17711  28657  46368  75025
121393  196418  317811  514229  832040
-----
Golden ratio convergence:
1.0  2.0  1.5  1.6666666666666667  1.6  1.625  1.6153846153846154  1.61904761904
29 1.6180339887482036  28 1.6180339887543225  27 1.618033988738303

 ----jGRASP: operation complete.
```

Golden Ratio

Fibonacci in Java

main

Lab 2

```
public class Fibs {
    static String spc2 = "   ";
    static int Ngen = 30, Nprt = 30, line = 12;
    public static void main(String[] args) {
        long[] F = new long[Ngen];
        //call Generate method
        genFibs(Ngen,F);
        //call Print method
        printit(Nprt,F);
        //call Golden method
        golden(Ngen,F);
    }//end main
```

Fibonacci in Java

Lab 2

Print inline

Print sub

```
20    //Generate method
21    static void genFibs(int N,long[] F) {
22       F[0]=1; F[1]=1;
23       System.out.println("Fibs as gen: ");
24       System.out.print(F[0] +spc2 +F[1] +spc2);
25       for(int i=2; i<N; i++) {
26          F[i] = F[i-1] + F[i-2];
27          if(i<=Nprt) System.out.print(F[i] +spc2);//max limit
28          if(i>0 && i<=Nprt && i%line==0) System.out.println();//15 per line
29          }//end for
30       //System.out.println("\nfinal 3: " +F[N-3] +spc2 +F[N-2] +spc2 +F[N-1]);
31       System.out.println("-----");
32    }//end Gen
33
34    //Print method
35    static void printit(int N, long[] F) {
36       System.out.println("Fibs from array: " +N);
37       for(int i=0;i<N;i++) {
38          if(i<=Nprt) System.out.print(F[i] +spc2);//max limit
39          if(i>0 && i%line==0) System.out.println();//15 per line
40          }
41       System.out.println("\n-----");
42    }//end print
```

Fibonacci in Java

Lab 2

```
44     //Golden method
45     static void golden(int N, long[] F) {
46        int M = 13;//1st 10 + last 3
47        double[] gold = new double [M];
48        System.out.println("Golden ratio convergence: ");
49        for(int i=0;i<10;i++) {
50           gold[i] = F[i+1]/(double)F[i];
51           System.out.print(gold[i] +spc2);}
52           System.out.println();
53        for(int i=0;i<3;i++) {
54           gold[i+3] = F[N-1-i]/(double)F[N-2-i];
55        System.out.print(N-1-i +" " +gold[i+3] +spc2);}
56        System.out.println();
57     }//end Golden
58  }//end class
```

Array Swap


```
//C conversion to MIPS
int N = 5; // some value
int a[ ] = {1,2,3,4,5}
int b[ ]= {6,7,8,9,10}
for (int i=0; I<N; I++) {//swap
   int temp = a[i];
   a[i] = b[i]
   b[i] = temp;
}
```

```
#MIPS by Jeff Drobman, 4-6-21
#reg map: $t0=count, $t1=*a, $t2=*b, $t3=temp
.eqv N, 5
.data
a:  .word 1,2,3,4,5
spacer: .ascii "****&&&&****" #3 words
b:  .word 6,7,8,9,10
.text
li $t0,N #loop counter
la $t1,a
la $t2,b

Loop: #for(I=N; I>0; I--)
lw $t3,($t1) #temp=a
lw $t4,($t2) #t4=b
sw $t4,($t1) #a=b
sw $t3,($t2) #b=temp
#update ptrs
addiu $t1,$t1,4
addiu $t2,$t2,4
#loop end
subi $t0,$t0,1
bgtz $t0,Loop #end Loop
#end of program
```

Array Swap

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	1	2	3	4	5	707406378	640034342	707406378
0x10010020	6	7	8	9	10	0	0	0

```
.data
a:   .word 1,2,3,4,5
spacer: .ascii "****&&&&****" #3 words
b:   .word 6,7,8,9,10
.text
li $t0,N #loop counter
la $t1,a
la $t2,b

Loop: #for(I=N; I>0; I--)
lw $t3,($t1) #temp=a
lw $t4,($t2) #t4=b
sw $t4,($t1) #a=b
sw $t3,($t2) #b=temp
#update ptrs
addiu $t1,$t1,4
addiu $t2,$t2,4
#loop end
subi $t0,$t0,1
bgtz $t0,Loop #end Loop
```

Swapped

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	6	7	8	9	10	707406378	640034342	707406378
0x10010020	1	2	3	4	5	0	0	0

Lab

LAB 7 MIPS

Factorials

Factorials

Lab 7

Java: non-*recursive*

```
import java.util.Scanner;

public class Facts {
   static String spc2 = "   ";
   static int Ngen = 30, Nprt = 30, line = 12;
   public static void main(String[] args) {
      long[] F = new long[Ngen];
      //get Ngen
      Scanner input = new Scanner(System.in);
      System.out.println("Input N: ");
      Ngen = input.nextInt();
      //call Generate method
      genFacts(Ngen,F);
      //call Print method
      if(Nprt>Ngen) Nprt = Ngen;//limit
      printit(Nprt,F);
   }//end main

   //Generate method
   static void genFacts(int N,long[] F) {
      F[0]=1;
      System.out.println("Factorials as gen: ");
      for(int i=1; i<N; i++) {
         F[i] = F[i-1] *i;
         if(i<=Nprt) System.out.print(F[i] +spc2);//max limit
         if(i>0 && i<=Nprt && i%line==0) System.out.println();//15 per line
         }//end for
      System.out.println("\n-----");
   }//end Gen
```

Factorials Results

Lab 7

Java Long 17

```
Input N:
17
Factorials as gen:
1  2  6  24  120  720  5040  40320  362880  3628800  39916800  479001600
6227020800  87178291200  1307674368000  20922789888000
-----
Factorials from array: 17
1  1  2  6  24  120  720  5040  40320  362880  3628800  39916800  479001600
6227020800  87178291200  1307674368000  20922789888000
--**--
```

```
Factorials by Jeff D
1 2 6 24 120 720 5040 40320 362880 3628800 39916800 479001600 1932053504 1278945280 2004310016 2004189184 -288522240
-- program is finished running --
```

MIPS assy

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	1952670022	1634300527	1646293868	1699356793	1142974054	1850277898	544503152	1769108595
0x10010020	3827566	1970302537	1919230068	561147762	0	1	2	6
0x10010040	24	120	720	5040	40320	362880	3628800	39916800
0x10010060	479001600	1932053504	1278945280	2004310016	2004189184	-288522240	0	0

Factorials Results

Lab 7

Java Long 17

```
Input N:
17
Factorials as gen:
1  2  6  24  120  720  5040  40320  362880  3628800  39916800  479001600
6227020800  87178291200  1307674368000  20922789888000
-----
Factorials from array: 17
1  1  2  6  24  120  720  5040  40320  362880  3628800  39916800  479001600
6227020800  87178291200  1307674368000  20922789888000
--**--
```

int 20

```
Input N:
20
Factorials as gen:
1  2  6  24  120  720  5040  40320  362880  3628800  39916800  479001600
1932053504  1278945280  2004310016  2004189184  -288522240  -898433024  109641728
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

Data, Macros & Code

Lab 7

```
## Lab 3 -- Factorials
## by Jeff Drobman
##version: 1.0 >12-12-19
.data
header: .asciiz  "Factorials by Jeff D\n"
prompt: .asciiz  "Input string:"
err_msg: .asciiz  "Input error!"
.align 2
facts: .word  0:12
size: .byte 0
#define
.eqv heap, 0x10040000
.eqv in_buf, 0x10040020 #input buffer
.eqv space, 0x20 #space
#.eqv heap, 0x10040000 -not used
#macros
.macro done
li $v0, 10 #stop code
syscall #stop
.end_macro
.macro GUI_out(%msg)
li $v0, 55 #code
la $a0, %msg
li $a1,1 #msg type is info
syscall
.end_macro
```

```
.text
##reg usage:
#t0=facts array ptr, t1=size decr N--
#t5=size (constant)
#s0=factors++ (1..N), s1=last factorial (i!)
jal GUI_in #get N
sb $a0,size #size=N
#set up pointers+counters
la $t0, facts #ptr+4
lbu $t5, size #N--
subu $t1,$t5,1 #counter-1
li $s0,1 #1st factor=1
li $s1,1 #1st factorial=1
sw $s1,($t0) #1st num=1
loop_main: #calc factorials->array
addiu $t0,$t0,4 #incr ptr+4
addiu $s0,$s0,1 #incr factor
multu $s0, $s1 #next product
mflo $s1
sw $s1,($t0) #->array
subi $t1,$t1,1 #counter-1
bgtz $t1,loop_main
#call sub for print
jal print
done #macro for exit
#---end of main---
```

Part A

Part B Use Subroutine

MIPS Multiply


```
mul $t1,$t2,$t3          Multiplication without overflow  : Set HI to high-order 32 bits, LO a
mul.d $f2,$f4,$f6        Floating point multiplication double precision : Set $f2 to double-pr
mul.s $f0,$f1,$f3        Floating point multiplication single precision : Set $f0 to single-pr
mult $t1,$t2             Multiplication : Set hi to high-order 32 bits, lo to low-order 32 bit
multu $t1,$t2            Multiplication unsigned : Set HI to high-order 32 bits, LO to low-orc
```

multu

```
mfhi $t1                 Move from HI register : Set $t1 to contents of HI (
mflo $t1                 Move from LO register : Set $t1 to contents of LO (
```

MIPS MULT

Multiply

`mult rs, rt`

0	rs	rt	0	0x18
6	5	5	10	6

Unsigned multiply

`multu rs, rt`

0	rs	rt	0	0x19
6	5	5	10	6

Multiply registers `rs` and `rt`. Leave the low-order word of the product in register `lo` and the high-order word in register `hi`.

Multiply (without overflow)

`mul rd, rs, rt`

0x1c	rs	rt	rd	0	2
6	5	5	5	5	6

Put the low-order 32 bit of the product of `rs` and `rt` into register `rd`.

Multiply (with overflow)

`mulo rdest, rsrc1, src2` *pseudoinstruction*

Unsigned multiply (with overflow)

`mulou rdest, rsrc1, src2` *pseudoinstruction*

Put the low-order 32 bits of the product of register `rsrc1` and `src2` into register `rdest`.

MIPS MULT

Multiply add

`madd rs, rt`

0x1c	rs	rt	0	0
6	5	5	10	6

Unsigned multiply add

`maddu rs, rt`

0x1c	rs	rt	0	1
6	5	5	10	6

Multiply registers `rs` and `rt` and add the resulting 64-bit product to the 64-bit value in the concatenated registers `lo` and `hi`.

Multiply subtract

`msub rs, rt`

0x1c	rs	rt	0	4
6	5	5	10	6

Unsigned multiply subtract

`msub rs, rt`

0x1c	rs	rt	0	5
6	5	5	10	6

Multiply registers `rs` and `rt` and subtract the resulting 64-bit product from the 64bit value in the concatenated registers `lo` and `hi`.

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

Lab 3: code

DR JEFF SOFTWARE
INDIE APP DEVELOPER

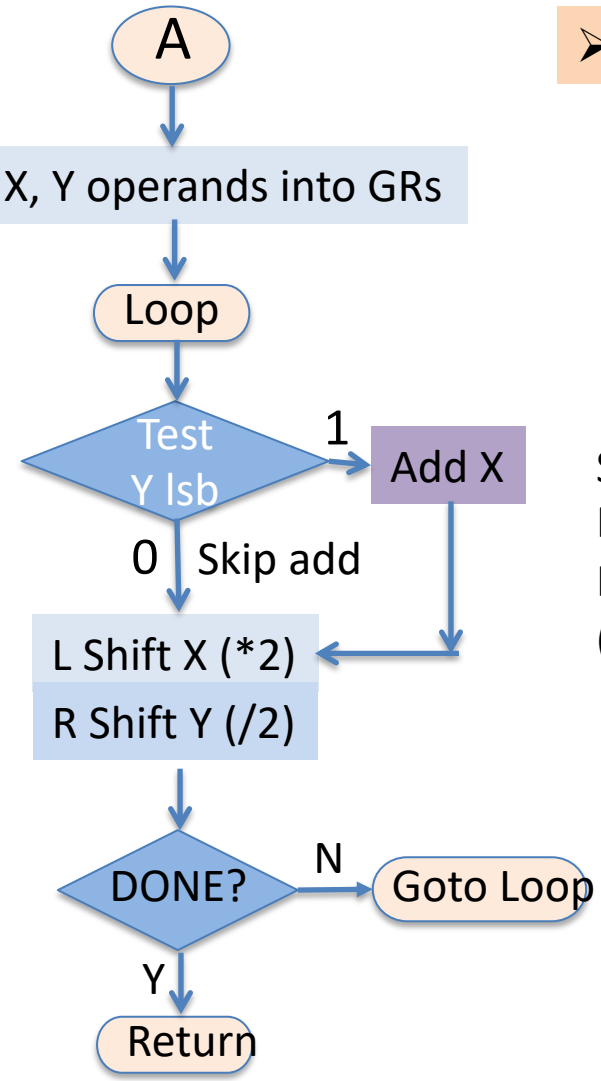
Lab 7

```
GUI_in: #sub-INput GUI msg
#a0=int, a1=status code
li $v0, 51 #int read
la $a0, prompt
syscall
bltz $a1, in_error
jr $ra
in_error:
    GUI_out(err_msg)
    b GUI_in
#subroutine: print($t0=ptr, $t5=size)
print: nop #sub label
#print header
li $v0, 4
la $a0, header
syscall
#print factorials
la $t0, facts #ptr
loop_prt:
li $v0, 1 #print int code
lw $a0, ($t0)
syscall
li $v0, 11 #print char code
li $a0, space
syscall
addi $t0,$t0,4 #incr ptr
subi $t5,$t5, 1 #count--
bgtz $t5,loop_prt
#return
jr $ra
#--end of program--
```

```
Factorials by Jeff D
1 2 6 24 120 720 5040 40320 362880 3628800 39916800 479001600
-- program is finished running --
```

Lab 7 Part B: Mult Sub

- Signed?



Hint: use mask

Mask: .word 0x0001

Summing Partial Products (bit-wise)

Binary Multiplication

Unsigned Multiplication

1 bit at a time

Y	Multiple	Op
0	0	none
1	1	add

2 bits at a time

Y	Multiple	Op
00	0	none
01	1	add
10	2	Shift-add
11	3	Add twice

Binary Multiplication

Signed 2'sC Multiplication

Drobman MS Thesis

Booth's Recoding

y_{i+1}	y_i	y_{i-1}	y^*_{i+1}	y^*_i	M_i	Operation
0	0	0	0	0	0	add 0
0	0	1	0	1	1	add X
0	1	0	0	1	1	add X
0	1	1	1	0	2	add 2X
1	0	0	$\bar{1}$	0	$\bar{2}$	subtract 2X
1	0	1	$\bar{1}$	1	$\bar{1}$	subtract X
1	1	0	0	$\bar{1}$	$\bar{1}$	subtract X
1	1	1	0	0	0	subtract 0

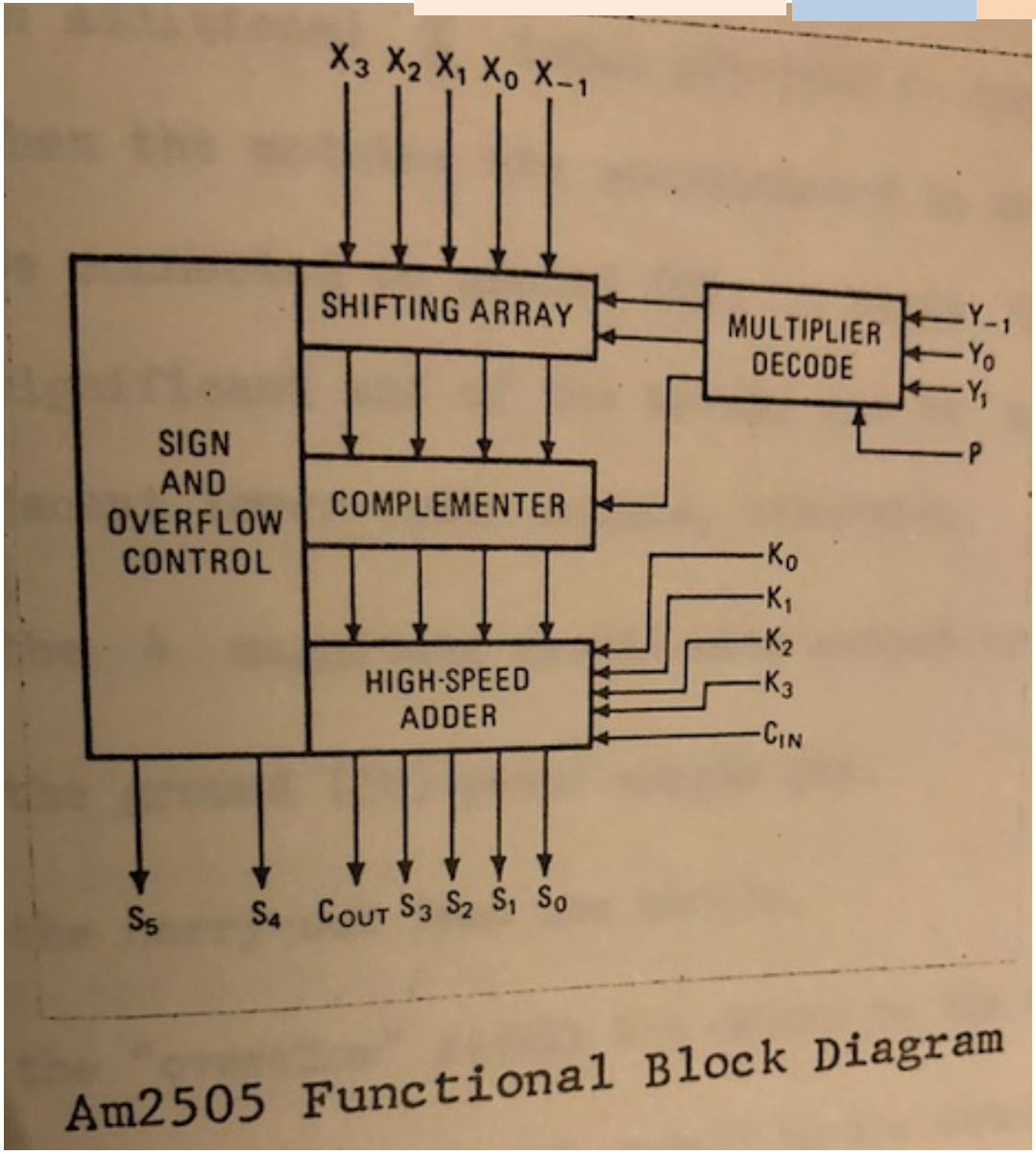
*bits recoded as a 1-string transformation

TABLE 2.1

Second-Order Recoding

Am2505 Multiplier

Drobman MS Thesis | Bit-slice | 1971-80



Am2505 Functional Block Diagram

Am2505 Multiplier

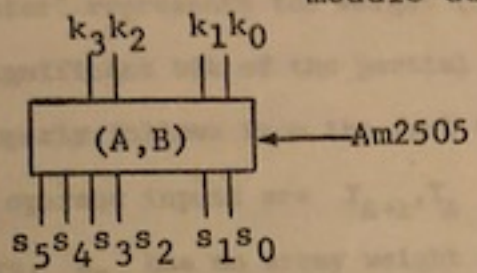
Drobman MS Thesis | Bit-slice | 1971-80

Notation: (from [1], [15], [16])

multiplier pair = Y_{A+1}, Y_A

multiplicand group = $X_{B+3}, X_{B+2}, X_{B+1}, X_B$

"module designator" = (A,B)



2x4-bit slices

8-bit x 8-bit multiply

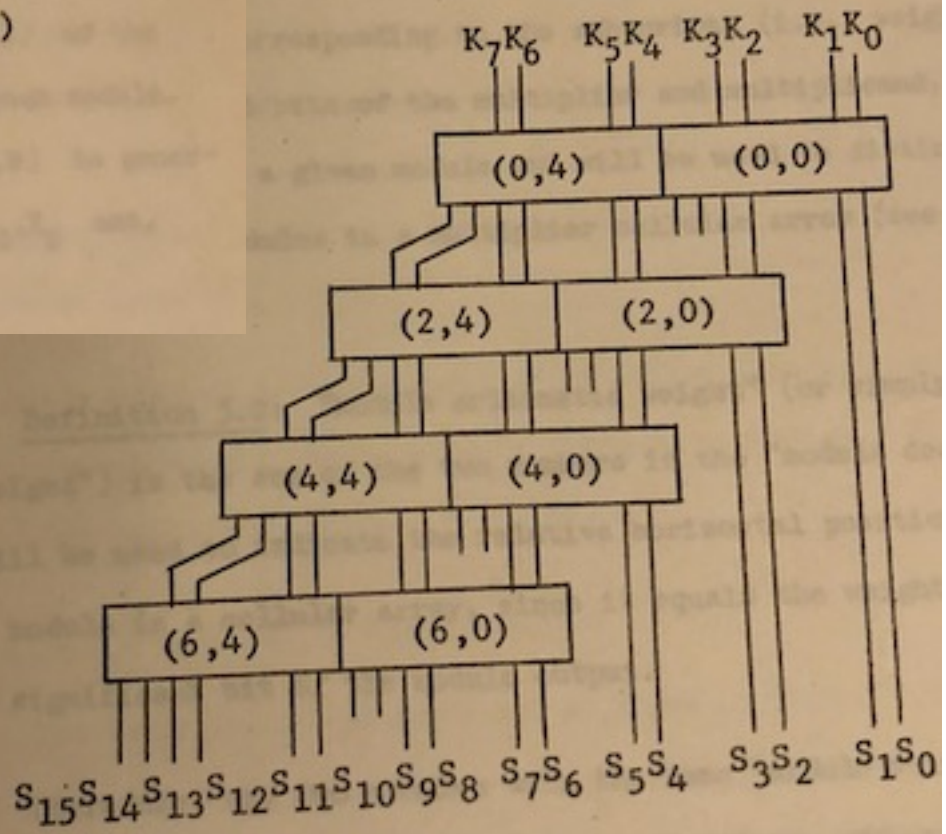


Figure 3.1. Basic Interconnection Scheme

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Lab 7 A & B: Print

DR JEFF SOFTWARE
INDIE APP DEVELOPER


```
Factorials by Jeff D
1 2 6 24 120 720 5040 40320
====
Factorials using B multiply
1 5 23 119 719 5039 40319 362879
-- program is finished running --
```

```
#call sub for print
la $a0, headerA
la $t0, facts #ptr
jal print
#repeat for B
la $a0, headerB
la $t0, factsB #ptr
jal print
done #macro for exit
#---end of main---
```

Lab 7 A & B: .data


```
## Lab 3 -- Factorials
## by Jeff Drobman
##version: 2.0 >11-16-20 Parts A & B
.data
headerA: .asciiz  "Factorials by Jeff D\n"
headerB: .asciiz  "\n====\nFactorials using B multiply\n"
prompt: .asciiz  "Input string:"
err_msg: .asciiz  "Input error!"
.align 2
alignA: .ascii "****Fact A*****==="
facts: .word  0:20
alignB: .ascii "****Fact B*****==="
factsB: .word  0:20 #B array
size: .byte 0
#define
```

Lab 7 A & B: Main


```
#t0, $t8, =facts array ptrs, t1=size decr N--
#t5=size (constant)
#s0=factors++ (1..N), t6,s6=last factorials
jal GUI_in #get N
sb $a0,size #size=N
#set up pointers+counters
la $t0, facts #ptr+4
la $t8, factsB #ptr+4
sub $t1,$a0,1 #counter-1
```

Exercise for the student

```
sw $s6,($t8) #->B array
subi $t1,$t1,1 #counter-1
bgtz $t1,loop_main
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

Lab 7 A & B: .data

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	t c a F	a i r o	b s l	e J y	D f f	= \n \0 \n	\n = = =	t c a F
0x10010020	a i r o	u s l	g n i s	m B	i t l u	\n y l p	p n I \0	s t u
0x10010040	n i r t	I \0 : g	t u p n	r r e	\0 ! r o	* * * *	t c a F	* * A
0x10010060	= = * *	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 x	\0 \0 . .	\0 \0 . .
0x10010080	\0 \0 . .	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	* * * *	t c a F	* * B
0x100100c0	= = * *	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 w	\0 \0 . .	\0 \0 . .	\0 \0 . .
0x100100e0	\0 . . .	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

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

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	1952670022	1634300527	1646293868	1699356793	1142974054	1024065546	171785533	1952670022
0x10010020	1634300527	1965060972	1735289203	1830830624	1769237621	175729776	1886275840	1931506805
0x10010040	1852404340	1224751719	1953853550	1920099616	2191983	707406378	1952670022	707412256
0x10010060	1027418666	1	2	6	24	120	720	5040
0x10010080	40320	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	707406378	1952670022	707412512
0x100100c0	1027418666	1	5	23	119	719	5039	40319
0x100100e0	362879	0	0	0	0	0	0	0

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

Factorials – Recursive

Lab 7

C or Java: *recursive*

```
//call Recursive method
System.out.println("Factorials-recursive: ");
int x = factRec(Ngen);
printit(Nprt,FR);
```

```
int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

```
51    //Recursive method
52    static int factRec(int N) {
53       System.out.print("FR:" +N +spc);
54       if(N==0) {
55          System.out.println();
56          return 1;}
57       //call recursively
58       FR[N] = N * factRec(N-1);//store
59       return FR[N];
60    }//end factRec
```

Store in array

Factorials – Recursive

Lab 7

Java: *recursive*

```
Input N:
17
Factorials as gen:
1  2  6  24  120  720  5040  40320  362880  3628800  39916800  479001600
1932053504  1278945280  2004310016  2004189184
-----
Factorials from array: 17
1  2  6  24  120  720  5040  40320  362880  3628800  39916800  479001600
1932053504  1278945280  2004310016  2004189184
------
Factorials-recursive:
FR:17 FR:16 FR:15 FR:14 FR:13 FR:12 FR:11 FR:10 FR:9 FR:8 FR:7 FR:6 FR:5 FR:4 FR:3
Factorials from array: 17
1  2  6  24  120  720  5040  40320  362880  3628800  39916800  479001600
1932053504  1278945280  2004310016  2004189184
```

2B

```
Factorials from array: 18
1  2  6  24  120  720  5040  40320  362880  3628800  39916800  479001600
1932053504  1278945280  2004310016  2004189184  -288522240
```

>>2B

Lab

LAB 9 MIPS

Interrupt Handlers

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Interrupts & Exceptions

P&H Ch 7

COMP 122: Computer Architecture and Assembly Language
Spring 2020

7.7 Exceptions and interrupts

(Original section[1])

COD Section 4.9 (Exceptions) describes the MIPS exception facility, which responds both to exceptions caused by errors during an instruction's execution and to external interrupts caused by I/O devices. This section describes exception and *interrupt handling* in more detail.[2] In MIPS processors, a part of the CPU called coprocessor 0 records the information the software needs to handle exceptions and interrupts. The MIPS simulator SPIM does not implement all of coprocessor 0's registers, since many are not useful in a simulator or are part of the memory system, which SPIM does not implement. However, SPIM does provide the following coprocessor 0 registers:

Figure 7.7.1: Coprocessor 0 registers.

Register name	Register number	Usage
BadVAddr	8	memory address at which an offending memory reference occurred
Count	9	timer
Compare	11	value compared against timer that causes interrupt when they match
Status	12	interrupt mask and enable bits
Cause	13	exception type and pending interrupt bits
EPC	14	address of instruction that caused exception
Config	16	configuration of machine

Feedback?

Interrupt handler: A piece of code that is run as a result of an exception or an interrupt.

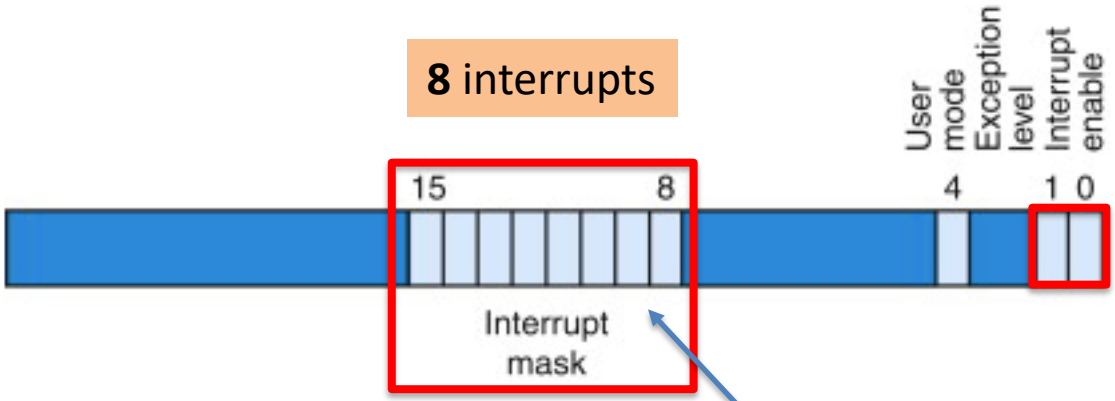
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Interrupts & Exceptions

P&H Ch 7

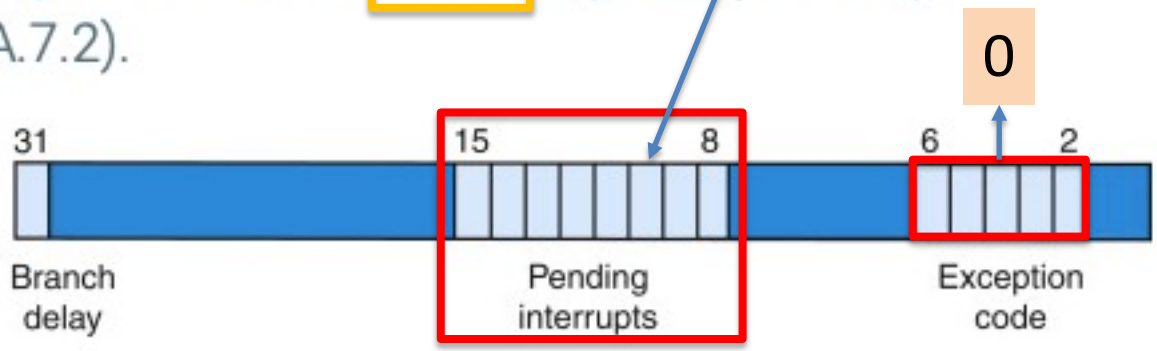
COMP 122: Computer Architecture and Assembly Language
Spring 2020

Figure 7.7.2: The status register (COD Figure A.7.1).



Mask(n) **&** Pending(n) → **INT(n)**

Figure 7.7.3: The cause register (COD Figure A.7.2).



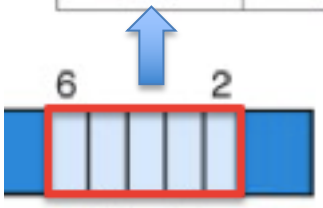
Interrupts & Exceptions

P&H Ch 7

COMP 122: Computer Architecture and Assembly Language
Spring 2020

Figure 7.7.4: Causes of exceptions.

	Number	Name	Cause of exception
0x00	0	Int	interrupt (hardware)
	4	AdEL	address error exception (load or instruction fetch)
	5	AdES	address error exception (store)
	6	IBE	bus error on instruction fetch
	7	DBE	bus error on data load or store
0x20	8	Sys	syscall exception
	9	Bp	breakpoint exception
	10	RI	reserved instruction exception
	11	CpU	coprocessor unimplemented
	12	Ov	arithmetic overflow exception
0x34	13	Tr	trap
	15	FPE	floating point



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff13
$13 (cause)	13	0x00000034
$14 (epc)	14	0x00400080

0x34

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Interrupts

General | MIPS

CLASSES

- **MASKABLE**
  - NMI (non-maskable)
    - Power-ON Reset
  - INT (maskable)
- **VECTORED**
  - NVI (non-V)
  - VI
- PRIORITY (PIC)
  - High
  - Low (High INTs "preempt" Low)
- INTERNAL
  - Hardware events
    - Timers
    - ADC
    - I/O (S, P)
  - Software exceptions

INT's (8)

- INT 0 (Pin 33)
- INT 1 (Pin 34)
- INT 2 (Pin 35)
- INT 7

ENABLES

- GIE (2) – global (2 groups)
- *Mask*: INT 0-7

PRIORITIES

- HIGH
- LOW

→ SAVED ON STACK

PROCESSOR STATE

- **PC**
- STATUS
- CAUSE

i8259 PIC

x86 IRQ's: 16 Intel

x86 IRQs [edit]

Typically, on systems using the Intel 8259 PIC, 16 IRQs are used. IRQs 0 to 7 are managed by one Intel 8259 PIC, and IRQs 8 to 15 by a second Intel 8259 PIC. The first PIC, the master, is the only one that directly signals the CPU. The second PIC, the slave, instead signals to the master on its IRQ 2 line, and the master passes the signal on to the CPU. There are therefore only 15 interrupt request lines available for hardware.

On newer systems using the Intel APIC Architecture, typically there are 24 IRQs available, and the extra 8 IRQs are used to route PCI interrupts, avoiding conflict between dynamically configured PCI interrupts and statically configured ISA interrupts. On early APIC systems with only 16 IRQs or with only Intel 8259 interrupt controllers, PCI interrupt lines were routed to the 16 IRQs using a PIR integrated into the southbridge.

The easiest way of viewing this information on Windows is to use Device Manager or System Information (msinfo32.exe). On Linux, IRQ mappings can be viewed by executing `cat /proc/interrupts` or using the `procinfo` utility.

Master PIC [edit]

- IRQ 0 – system timer (cannot be changed)
- IRQ 1 – keyboard controller (cannot be changed)
- IRQ 2 – cascaded signals from IRQs 8–15 (any devices configured to use IRQ 2 will actually be using IRQ 9)
- IRQ 3 – serial port controller for serial port 2 (shared with serial port 4, if present)
- IRQ 4 – serial port controller for serial port 1 (shared with serial port 3, if present)
- IRQ 5 – parallel port 2 and 3 or sound card
- IRQ 6 – floppy disk controller
- IRQ 7 – parallel port 1. It is used for printers or for any parallel port if a printer is not present. It can also be potentially be shared with a secondary sound card with careful management of the port.

Slave PIC [edit]

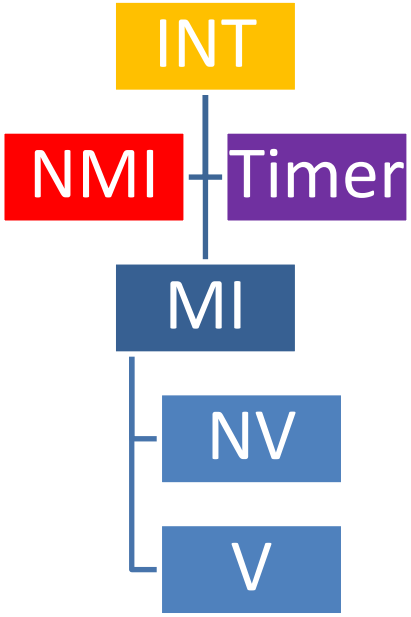
- IRQ 8 – real-time clock (RTC)
- IRQ 9 – Advanced Configuration and Power Interface (ACPI) system control interrupt on Intel chipsets.[2] Other chipset manufacturers might use another interrupt for this purpose, or make it available for the use of peripherals (any devices configured to use IRQ 2 will actually be using IRQ 9)
- IRQ 10 – The Interrupt is left open for the use of peripherals (open interrupt/available, SCSI or NIC)
- IRQ 11 – The Interrupt is left open for the use of peripherals (open interrupt/available, SCSI or NIC)
- IRQ 12 – mouse on PS/2 connector
- IRQ 13 – CPU co-processor or integrated floating point unit or inter-processor interrupt (use depends on OS)
- IRQ 14 – primary ATA channel (ATA interface usually serves hard disk drives and CD drives)
- IRQ 15 – secondary ATA channel

Lab 9 INT Model

INT's Used: 4

- **MASKABLE (3)**
  - **1** NMI (non-maskable)
    - Power-ON Reset
  - **2** INT (maskable)
- **VECTORED (2)**
  - **1** NVI (non)
  - **1** VI
- **TIMER (1)**

Hierarchy: ***Priority***



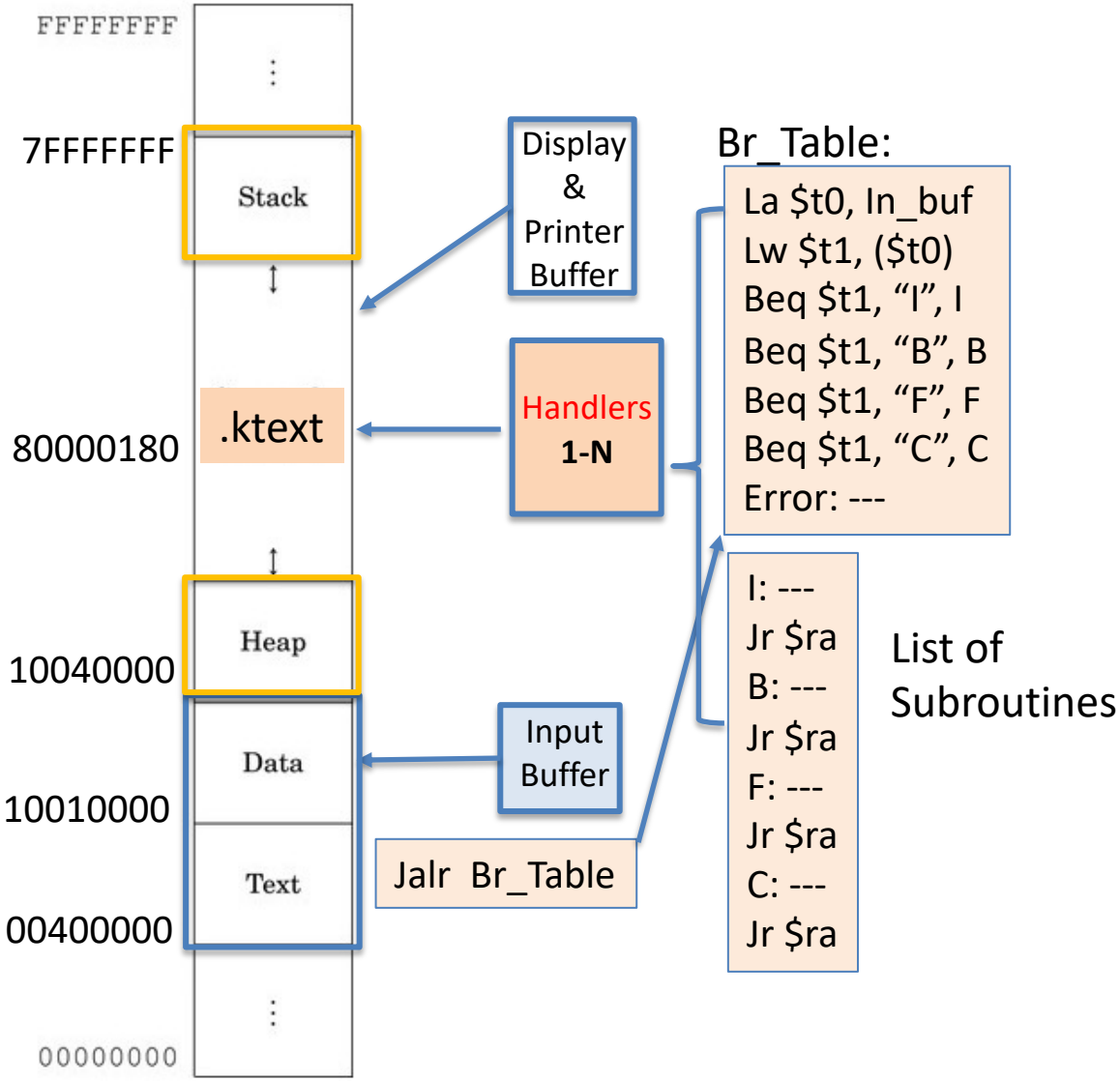
Interrupt Handlers

Lab 9

- Decode *Pending* Interrupts
- Allocate memory for Handlers
- Use *Jump Table*
  - Order by *Priority*
  - Test & Jump
  - Handlers as subroutines: jal → jr $ra

Lab 4: Memory

Lab 9



Typical memory layout for a program with a 32-bit address space.

MIPS Memory Config

Lab 9

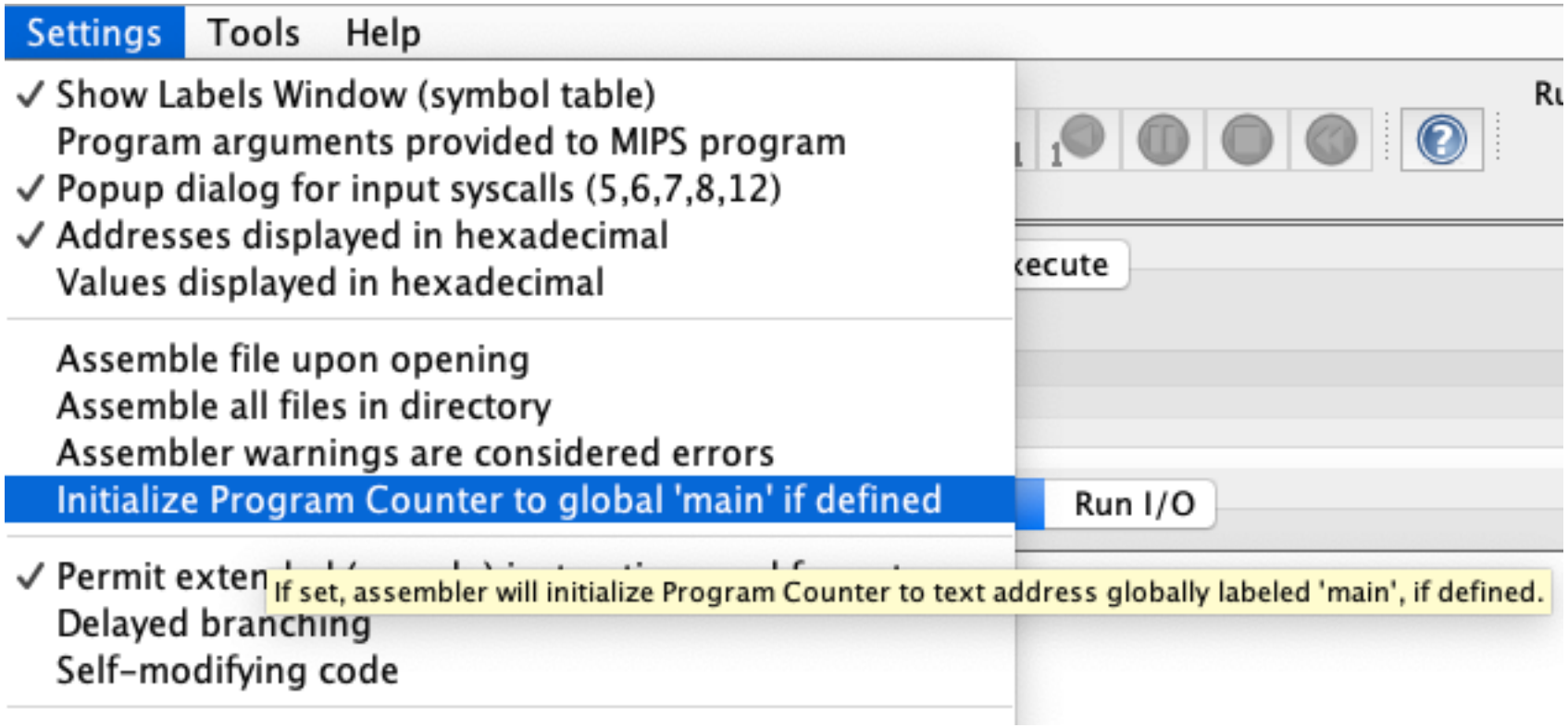
MIPS Memory Configuration

0xffffffff	memory map limit address
0xffffffff	kernel space high address
0xffff0000	MMIO base address
0xfffeffff	kernel data segment limit address
0x90000000	.kdata base address
0x8ffffffc	kernel text limit address
0x80000180	exception handler address
0x80000000	kernel space base address
0x80000000	.ktext base address
0x7fffffff	user space high address
0x7fffffff	data segment limit address
0x7ffffffc	stack base address
0x7fffeffc	stack pointer $sp
0x10040000	stack limit address
0x10040000	heap base address
0x10010000	.data base address
0x10008000	global pointer $gp
0x10000000	data segment base address
0x10000000	.extern base address
0x0ffffffc	text limit address
0x00400000	.text base address

ess 0
ess 0

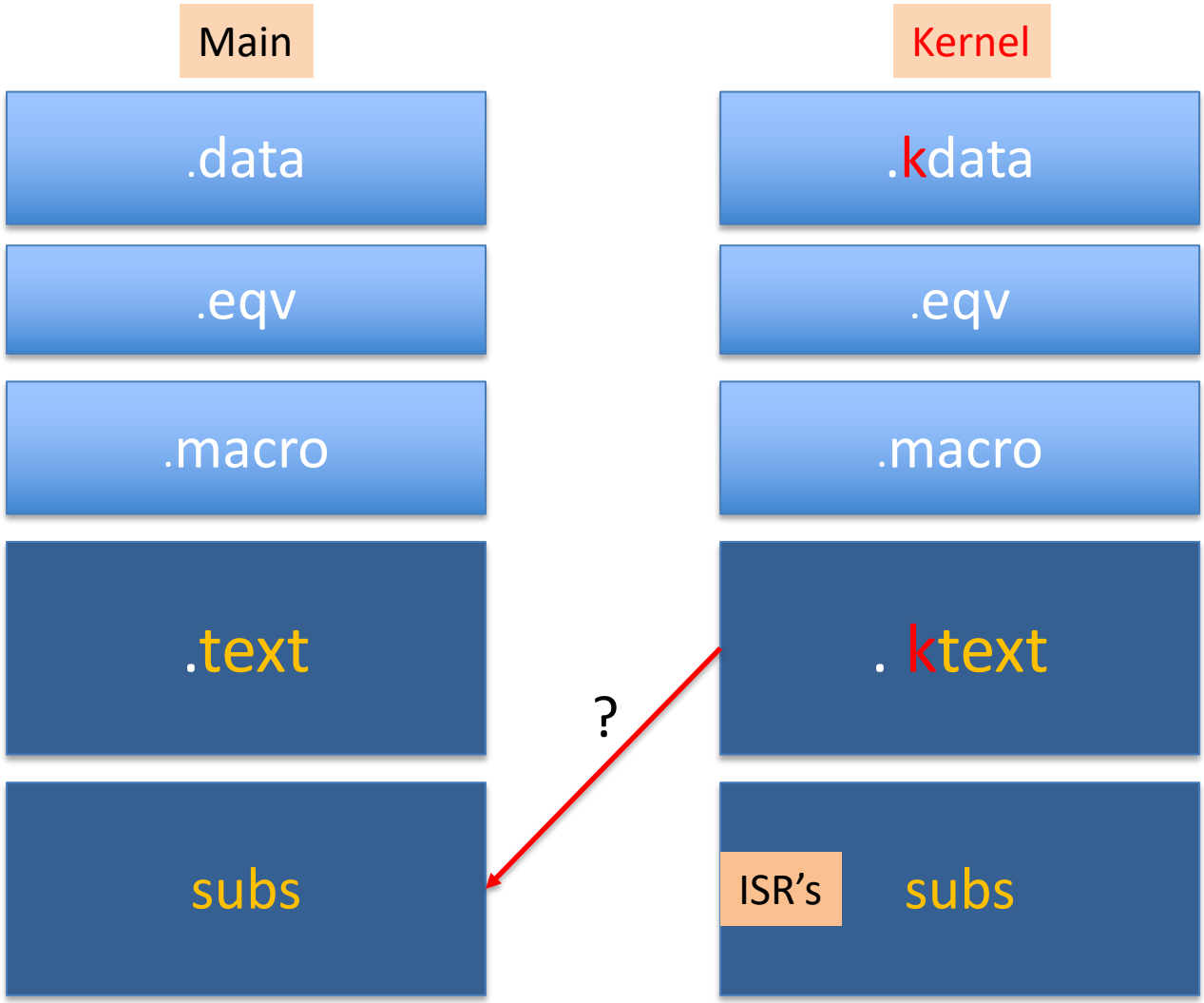
Global Main

#ifDef



Lab 9 Program Structure

Lab 9



Lab 9 Program

Lab 9

```
## Lab 4 -- Interrupts
## by Jeff Drobman
##version: 1.0 >4.14.20
#Interrupt vectors:
# 0-3 active
.data
vector: .ascii "#cev" #reserve 4 bytes
header: .asciiz  "Lab 4: Interrupts by Jeff D\n"
.align 2
prompt1: .asciiz "Enter Int TYPE: 0=NMI, 1=NVI, 2=VI, 9=Halt"
.align 2
prompt2: .asciiz "Enter Int Vector (0-15):"
.align 2
NMI_str: .asciiz "NMI interrupt!"
.align 2
NVI_str: .asciiz "NVI interrupt!"
.align 2
VI_str: .asciiz "Vectored interrupt!"
.align 2
Err_msg: .asciiz "Error: illegal Int Type"
.align 2
Halt_msg: .asciiz "Halted! Good-bye"
.align 2
Stop_msg: .asciiz "Stopped out!"
.align 2
```

Lab 9 Code Macros


```
#define
.eqv heap, 0x10040000
.eqv in_buf, 0x10040020 #input buffer
.eqv exc_seg, 0x80000180
.eqv stop,5
#macros
.macro done
li $v0, 10 #stop code
syscall #stop
.end_macro
.macro print_mac (%str)
la $a0, %str
li $v0, 4
syscall
.end_macro
.macro msgbox (%str)
la $a0, %str
li $v0, 55 #GUI msg code
li $a1, 1 #msg type is info
syscall
.end_macro
```

```
#**ISR macro->Trap
.macro _ISR (%str)
la $a0,%str
jal GUI_out
Teq $0,$0 #Trap: simulate INT<-1 (in ktext)
b loop_main
.end_macro
```

Lab 9 Code Loop

Lab 9

```
loop_main:
    subiu $t9,$t9,1 #decr counter
    blez $t9,Stop
    la $a0,prompt1
    Jal GUI_in #get Type in $a0
    #Int TYPE Branch table (if-case)
    beq $a0,0,NMI
    beq $a0,1,NVI
    beq $a0,2,VI
    beq $a0,9,Halt
    b Err #none of above
    NMI: _ISR(NMI_str)
    NVI: _ISR(NVI_str)
    VI:  #get vector
        la $a0,prompt2
        Jal GUI_in #get vector in $a0
        _ISR(VI_str)
    Halt:subiu $a0,$a0,7 #chk for 9
         bltz $a0,Err #<9
         la $a0,Halt_msg
         jal GUI_out
         jal printStr
         done #**exit program**
    Err: la $a0,Err_msg1 #default
         jal GUI_out
```

"_ISR" macro

Lab 9 Subs


```
#--END main Loop--
##subroutines follow**
#print $a0 on console
printStr:
li $v0, 4
syscall
jr $ra
#OUTput GUI msg
GUI_out: #ptr in $a0
li $v0, 55 #GUI msg code
li $a1, 1 #msg type is info
syscall
jr $ra
#INput GUI msg
GUI_in:
li $v0, 54 #GUI msg code
la $a1, in_buf
li $a2, 4 #max input length
syscall
Lbu $a0, in_buf #return 1st byte
jr $ra
```

Lab 9 Kernel Code/Data


```
.kdata
kmsg: .asciiz "starting Interrupt handler...\n"
def_msg: .asciiz "Error: unimplemented vector\n"
.align 2
end_Kdata: .asciiz "ENDkDATA$$$$"

.ktext exc_seg
#save state
push_k
print_mac kmsg #prt msg via macro
mfc0 $t0, $14 #EPC
addi $t0,$t0, 4 #incr RA in EPC
mtc0 $t0, $14 #EPC+4 (for ERET)
#--Branch Table--
Beq $a0, 0, v0
Beq $a0, 1, v1
Beq $a0, 2, v2
Beq $a0, 3, v3
#default
b def
#end Br table
#--Vector Table--
v0: #ISR for v0
```

```
#--end subs--
#**start handler code in kernel seg**
.macro push_k
move $k0, $v0 #save regs
move $k1, $a0
.end_macro
.macro pop_k
move $v0, $k0 #restore regs
move $a0, $k1
eret
.end_macro
```

```
#--Vector Table--
v0: #ISR for v0
v1:
v2:
v3:
def: #un-impl
print_mac def_msg
```

Lab 9 Kernel Code

macros

```
#**start handler code in kernel seg**
.macro push_k
move $k0, $a0 #save regs
move $k1, $a1
.end_macro
.macro pop_k
move $a0, $k0 #restore regs
move $a1, $k1
.end_macro
.macro _print %str
la $a0,%str
jal print_str
b return
.end_macro
```

subs

```
#--end ISR's--
#--subs--
print_str: #$a0=string ptr
    li $v0,4
    syscall
    b newline
print_val: #$a0=val
    li $v0,1 #int
    syscall
    #fall thru
newline:
    li $v0,4
    la $a0,newLn #"\n"
    syscall
    jr $ra
#--delay loop--
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Interrupts & Exceptions

DR JEFF SOFTWARE
INDIE APP DEVELOPER

P&H Ch 7

COMP 122: Computer Architecture and Assembly Language
Spring 2020

```
.ktext 0x80000180
mov $k1, $at     # Save $at register
sw  $a0, save0   # Handler is not re-entrant and can't use
sw  $a1, save1   # stack to save $a0, $a1
                 # Don't need to save $k0/$k1
```

```
mfc0 $k0, $13          # Move Cause into $k0

srl  $a0, $k0, 2       # Extract ExcCode field
andi $a0, $a0, 0xf

bgtz $a0, done         # Branch if ExcCode is Int (0)

mov  $a0, $k0          # Move Cause into $a0
mfco $a1, $14          # Move EPC into $a1
jal  print_excp        # Print exception error message
```

Interrupts & Exceptions

P&H Ch 7

COMP 122: Computer Architecture and Assembly Language
Spring 2020

```
done:   mfc0    $k0, $14        # Bump EPC
        addiu   $k0, $k0, 4     # Do not re-execute
                                # faulting instruction
        mtc0    $k0, $14        # EPC

        mtc0    $0, $13         # Clear Cause register

        mfc0    $k0, $12        # Fix Status register
        andi    $k0, 0xfffd     # Clear EXL bit
        ori     $k0, 0x1        # Enable interrupts
        mtc0    $k0, $12

        lw      $a0, save0      # Restore registers
        lw      $a1, save1
        mov     $at, $k1

        eret                    # Return to EPC

        .kdata
save0:  .word 0
save1:  .word 0
```

Exception Handler

P&H Ch 5

Figure 5.7.10: MIPS code to save and restore state on an exception (COD Figure 5.34).

Save state			
Save GPR	addi	$k1,$sp, -XCPSIZE	# save space on stack for state
	sw	$sp, XCT_SP($k1)	# save $sp on stack
	sw	$v0, XCT_V0($k1)	# save $v0 on stack
	...		# save $v1, $a1, $s1, $t1,... on stack
	sw	$ra, XCT_RA($k1)	# save $ra on stack
Save hi, lo	mfhi	$v0	# copy Hi
	mflo	$v1	# copy Lo
	sw	$v0, XCT_HI($k1)	# save Hi value on stack
	sw	$v1, XCT_LO($k1)	# save Lo value on stack
Save exception registers	mfc0	$a0, $cr	# copy cause register
	sw	$a0, XCT_CR($k1)	# save $cr value on stack
	...		# save $v1,....
	mfc0	$a3, $sr	# copy status register
	sw	$a3, XCT_SR($k1)	# save $sr on stack
Set sp	move	$sp, $k1	# sp = sp - XCPSIZE
Enable nested exceptions			
	andi	$v0, $a3, MASK1	# $v0 = $sr & MASK1, enable exceptions
	mtc0	$v0, $sr	# $sr = value that enables exceptions

Exception Handler

P&H Ch 5

COMP 122: Computer Architecture and Assembly Language
Spring 2020

Call C exception handler			
Set $gp	move	$gp, GPINIT	# set $gp to point to heap area
Call C code	move	$a0, $sp	# arg1 = pointer to exception stack
	jal	xcpt_deliver	# call C code to handle exception
Restoring state			
Restore most GPR, hi, lo	move	$at, $sp	# temporary value of $sp
	lw	$ra, XCT_RA($at)	# restore $ra from stack
	...		# restore $t0,....., $a1
	lw	$a0, XCT_A0($k1)	# restore $a0 from stack
Restore status register	lw	$v0, XCT_SR($at)	# load old $sr from stack
	li	$v1, MASK2	# mask to disable exceptions
	and	$v0, $v0, $v1	# $v0 = $sr & MASK2, disable exceptions
	mtc0	$v0, $sr	# set status register
Exception return			
Restore $sp and rest of GPR used as temporary registers	lw	$sp, XCT_SP($at)	# restore $sp from stack
	lw	$v0, XCT_V0($at)	# restore $v0 from stack
	lw	$v1, XCT_V1($at)	# restore $v1 from stack
	lw	$k1, XCT_EPC($at)	# copy old $epc from stack
	lw	$at, XCT_AT($at)	# restore $at from stack
Restore ERC and return	mtc0	$k1, $epc	# restore $epc
	eret	$ra	# return to interrupted instruction

DEC PDP-11

1970 Wiki

No dedicated I/O instructions [edit]

Early models of the PDP-11 had no dedicated bus for input/output, but only a system bus called the Unibus, as input and output devices were mapped to memory addresses.

An input/output device determined the memory addresses to which it would respond, and specified its own interrupt vector and interrupt priority. This flexible

Interrupts [edit]

The PDP-11 supports hardware interrupts at four priority levels. Interrupts are serviced by software service routines, which could specify whether they themselves could be interrupted (achieving interrupt nesting). The event that causes the interrupt is indicated by the device itself, as it informs the processor of the address of its own interrupt vector.

Interrupt vectors are blocks of two 16-bit words in low kernel address space (which normally corresponded to low physical memory) between 0 and 776. The first word of the interrupt vector contains the address of the interrupt service routine and the second word the value to be loaded into the PSW (priority level) on entry to the service routine.

Instruction set orthogonality [edit]

See also: PDP-11 architecture

The PDP-11 processor architecture has a mostly orthogonal instruction set. For example, instead of instructions such as *load* and *store*, the PDP-11 has a *move* instruction for which either operand (source and destination) can be memory or register. There are no specific *input* or *output* instructions; the PDP-11 uses memory-mapped I/O and so the same *move* instruction is used; orthogonality even enables moving data directly from an input device to an output device. More complex instructions such as *add* likewise can have memory, register, input, or output as source or destination.

Most operands can apply any of eight addressing modes to eight registers. The addressing modes provide register, immediate, absolute, relative, deferred (indirect), and indexed addressing, and can specify autoincrementation and autodecrementation of a register by one (byte instructions) or two (word instructions). Use of relative addressing lets a machine-language program be position-independent.

DEC PDP-11

1st **LSI**-chip Computer | 1970 | Wiki

PDP-11/40. The processor is at the bottom. A TU56 dual DECtape drive is installed above it.

LAB: Interrupts

- Part 1 – Interrupts

  New code.c

  - Write C code to configure INT0 (High) and INT1 (Low)

- Part 2 – Timers

  Add to *old* hello.c

  - Write *DELAY* sub in C (2ms)
  - Use Timer 0 for new *DELAY* sub (2ms)
  - Write programs for:
    - Timer
    - Stopwatch (Counter)

LAB: Timers

- Part 1 – Finish assembly/C programs for "Hello World"
  - Assembly
    - Write *DELAY* subroutine
  - C
    - Use (call) *DELAY*
    - Output message via **Port B** (same)

- Part 2 – Timers
  - Write *DELAY* sub in C
  - Use Timer 0 for new *DELAY* sub
  - Write programs for:
    - Timer
    - Stopwatch (Counter)

Lab

Project A

MIPS vs ARM ISA's

Project A: MIPS ISA

MIPS 32

- List all these <u>instructions</u> in *MIPS32* (as in MARS)
  - ❑ Loads
  - ❑ Branches

➢ Primitives & Pseudos* (flag with asterisk)

Loads:
lw, lui, …

- MOV
- Reg-Mem
  - LOAD
  - STORE
  - MOV
- JUMP/
- BRANCH
  - BRA
  - BRCC
  - LOOP

Branches:
b, beq, …

➢ Note: both these classes involve an ***Effective Address (EA)*** calculation

Register (as base pointer) + ***Offset*** (as index)

Performed automatically

Project A: MIPS ISA

PROJECT FORM
Project 1

Requirements for **Part A**

- List all these instructions in *MIPS32* (as in MARS)
  - ❑ Loads
    Primitives:
    Pseudos:
  - ❑ Branches
    Primitives:
    Pseudos:

Include instruction *mnemonic* ("lw"),
but don't need to add all the EA format options ("100($t2) + label").

Project B: ISA's

MIPS vs ARM

1. Compare ISA's of MIPS and ARM by instruction <Class.sub.instr>: ex. <ALU.add>. Use the given slide as the baseline by identifying <u>missing</u> and <u>extra</u> instructions.

2. List instruction <u>Formats</u> (R, I, J) for each instruction class

3. Compare "Load" instructions of MIPS and ARM in <u>detail</u> (explain what each instruction does, what *modifiers* are used, and whether it is a *primitive* or *pseudo-op*:

MIPS: lb, lh, lw, li*
ARM: ??

Project B: #3

MIPS:

- Lb: Means to Load a byte (8 bits) on lower part from designated address to a selected register. This is a primitive.
- Lh: Means to Load a lower half word(16bits) from designated address to a selected register. This is a primitive.
- Lw: Means to Load a word(32bits) from designated address to a selected register. This is a primitive.
- La: Means to Load a designated address to a selected register. This is a pseudo-op.
- Li: Means to Load a stated value you entered in the code to a selected register. This is a pseudo-op.
- lui: Means to Load an upper half word(16bits) from value stated in code to a selected register's upper half. This is a primitive.

ARM:

- ldm: loads multiple registers into the program stack and has 8 different variation in its use.

ia	Increment After	ea	Empty Ascending
ib	Increment Before	fa	Full Ascending
da	Decrement After	ed	Empty Descending
db	Decrement Before	fd	Full Descending

❖ LDM

This is a pseudo-op.

- ldr: Loads contents to a single register from memory address and it has 5 variations (word, half word unsigned, byte unsigned , half word signed, byte signed) of content length. This is a primitive.
- ldr: The same as ldr, but the memory address can be an immediate value. This is a pseudo-op.
- ldrex: loads the contents of memory address into register and tags the memory address as true. This is a pseudo-op.

Lab

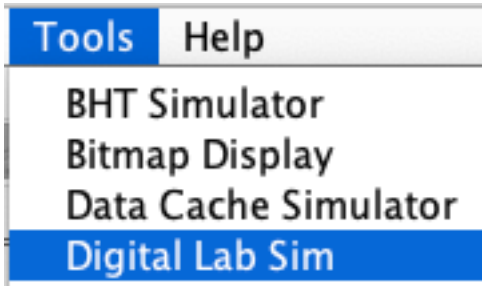
Project 1

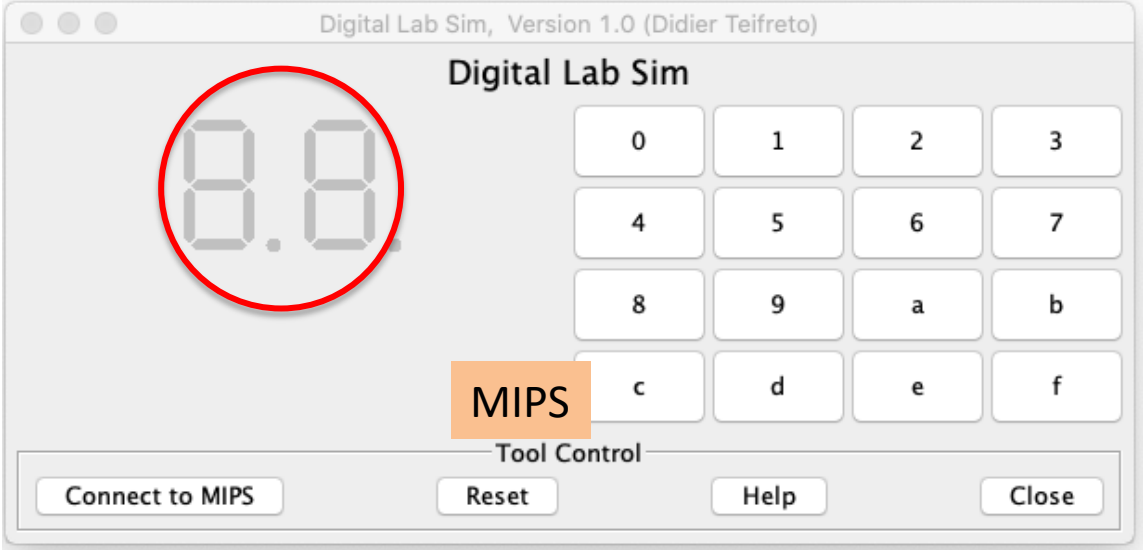
Digital Logic Lab (LED's)

MIPS

Project 1

- Logic simulators
  - MIPS MIPS
  - ARM
    - Embest board





Project 1

- Logic simulators
  - MARS *Digital Lab Sim* tool

    MIPS
- Functions (on LEDs)
  - I=Initial(s)
  - B=Blink (*all* on/off)
  - F=Flash (Initials-*sequential* on/off)
  - Z=Zero Flash (0-*sequential* on/off)
  - C=Counter
  - Q=Quit
  - Calculator (hex keypad – bonus)
- Use *Jump Table*
  - Input letter for Command
  - Test & Branch (beq)
  - Add Handlers as subroutines

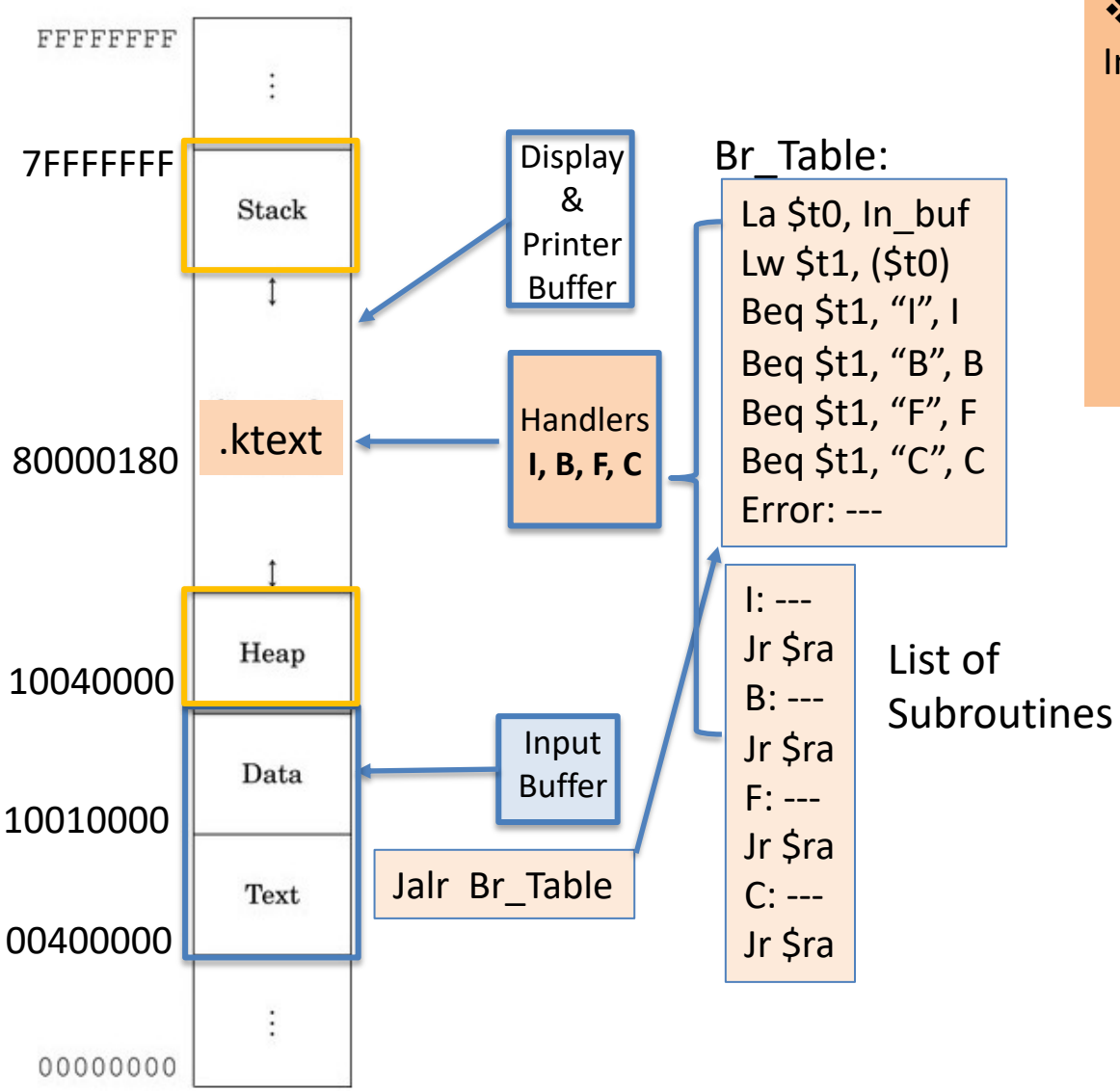
- Command Interpreter
  - I
  - B
  - F
  - Z
  - C
  - Q

Lab 1C

What's this? A ***Branch Table***

```
#checks input
check: nop
beq $a1, -2, cancel
beq $a1, -3, nodata
beq $a1, -4, exceed
jr $ra
```

Project 1: Memory



Typical memory layout for a program with a 32-bit address space.

- Command Interpreter
  - I
  - B
  - F
  - Z
  - [C] optional
  - Q

Table 1-3 ASCII Conver

Hex	Character
41	A
42	B
43	C
44	D
45	E
46	F
47	G
48	H
49	I
51	Q

Project 1

Writing and Using MIPS exception handlers in MARS

Introduction

Exception handlers, also known as *trap handlers* or *interrupt handlers*, can easily be incorporated into a MIPS program. This guide is not intended to be comprehensive but provides the essential information for writing and using exception handlers.

Although the same mechanism services all three, *exceptions*, *traps* and *interrupts* are all distinct from each other. Exceptions are caused by exceptional conditions that occur at runtime such as invalid memory address references. Traps are caused by instructions constructed especially for this purpose, listed below. Interrupts are caused by external devices.

MARS partially but not completely implements the exception and interrupt mechanism of SPIM.

Essential Facts

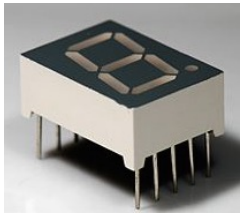
Some essential facts about writing and using exception handlers include:

- MARS simulates basic elements of the MIPS32 exception mechanism.
- If there is no instruction at location `0x800000180`, MARS will terminate the MIPS program with an appropriate error message.
- The exception handler can return control to the program using the `eret` instruction. This will place the EPC register $14 value into the Program Counter, so be sure to increment $14 by 4 before returning to skip over the instruction that caused the exception. The `mfc0` and `mtc0` instructions are used to read from and write to Coprocessor 0 registers.
- Bits 8-15 of the Cause register $13 can also be used to indicate pending interrupts. Currently this is used only by the Keyboard and Display Simulator Tool, where bit 8 represents a keyboard interrupt and bit 9 represents a display interrupt. For more details, see the Help panel for that Tool.
- Exception types declared in `mars.simulator.Exceptions`, but not necessarily implemented, are ADDRESS_EXCEPTION_LOAD (4), ADDRESS_EXCEPTION_STORE (5), SYSCALL_EXCEPTION (8), BREAKPOINT_EXCEPTION (9), RESERVED_INSTRUCTION_EXCEPTION (10), ARITHMETIC_OVERFLOW_EXCEPTION (12), TRAP_EXCEPTION(13), DIVIDE_BY_ZERO_EXCEPTION (15), FLOATING_POINT_OVERFLOW (16), and FLOATING_POINT_UNDERFLOW (17).
- When writing a non-trivial exception handler, your handler must first save general purpose register contents, then restore them before returning.

Project

APPLICATION – **LCD/LED Modules**

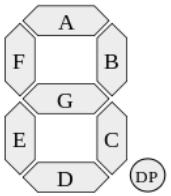
send letters/numbers to a 7-segment LCD



- Print your initials

J → BCD = 00001110 = 0x0E

- *Optional* extras
  - Calculator with Hex keys



Hexadecimal encodings for displaying the digits 0 to F

Digit	gfedcba	abcdefg	a	b	c	d	e	f	g
0	0x3F	0x7E	on	on	on	on	on	on	off
1	0x06	0x30	off	on	on	off	off	off	off
2	0x5B	0x6D	on	on	off	on	on	off	on
3	0x4F	0x79	on	on	on	on	off	off	on
4	0x66	0x33	off	on	on	off	off	on	on
5	0x6D	0x5B	on	off	on	on	off	on	on
6	0x7D	0x5F	on	off	on	on	on	on	on
7	0x07	0x70	on	on	on	off	off	off	off
8	0x7F	0x7F	on	on	on	on	on	on	on
9	0x6F	0x7B	on	on	on	on	off	on	on
A	0x77	0x77	on	on	on	off	on	on	on
b	0x7C	0x1F	off	off	on	on	on	on	on
C	0x39	0x4E	on	off	off	on	on	on	off
d	0x5E	0x3D	off	on	on	on	on	off	on
E	0x79	0x4F	on	off	off	on	on	on	on
F	0x71	0x47	on	off	off	off	on	on	on

Project 1 Code: Main

.data

```
## Project 2 -- Digital Lab
## by Jeff Drobman
##version: 1.1 >12-5-19
.data
header: .asciiz  "Project 2:  Digital Lab by Jeff D\n"
.align 2
prompt: .asciiz "Enter command:  I/B/F/C/Q"
.align 2
Com_str: .ascii "Now performing command: "
filler: .asciiz "*\0**" #I/B/F/C/Q +null goes in byte0-1
.align 2
Stop_msg: .asciiz "Stopped out!"
Quit_msg: .asciiz "user Quit"
newline:  .byte 0x0A
space:  .byte  0x20 #space
#define
```

Command will go here

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	j o r P	t c e	: 2	i g i D	l a t	b a L	J y b	f f e
0x10010020	\0 \0 \n D	e t n E	o c r	n a m m	: d	/ B / I	/ C / F	\0 \0 \0 Q
0x10010040	w o N	f r e p	i m r o	c g n	a m m o	: d n	* * \0 *	\0 \0 \0 \0
0x10010060	p o t S	d e p	! t u o	e s u \0	u Q r	\n \0 t i	\0 \0 \0	\0 \0 \0 \0
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

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

Proj 1: Macros & Defines

.data

```
#macros
.macro done
li $v0, 10 #stop code
syscall #stop
.end_macro
.macro print_mac (%str)
la $a0, %str
li $v0, 4
syscall
.end_macro
.macro msgbox (%str)
la $a0, %str
li $v0, 55 #GUI msg code
li $a1, 1 #msg type is info
syscall
.end_macro
.macro getChar
 Lbu $a0, in_buf #load char
 .end_macro
```

.macro

```
#define
.eqv heap, 0x10040000
.eqv in_buf, 0x10040020 #input buffer
.eqv hdlr_seg, 0x80000180
.eqv I, 0x49
.eqv Bx, 0x42
.eqv F, 0x46
.eqv C, 0x43
.eqv Q, 0x51
.eqv stop, 3
#macros
.macro done
```

.eqv

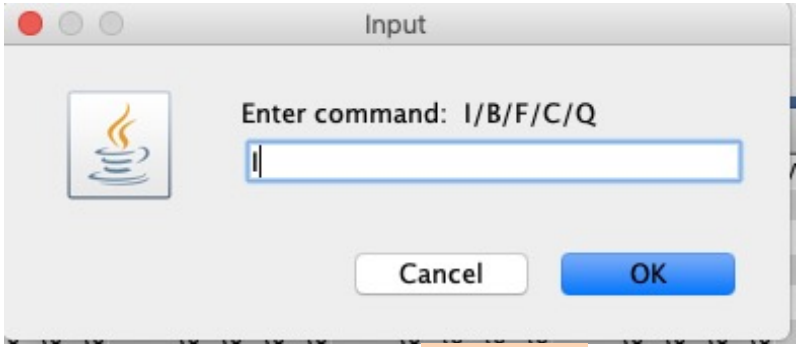
Adjust as needed

Project Code: Main

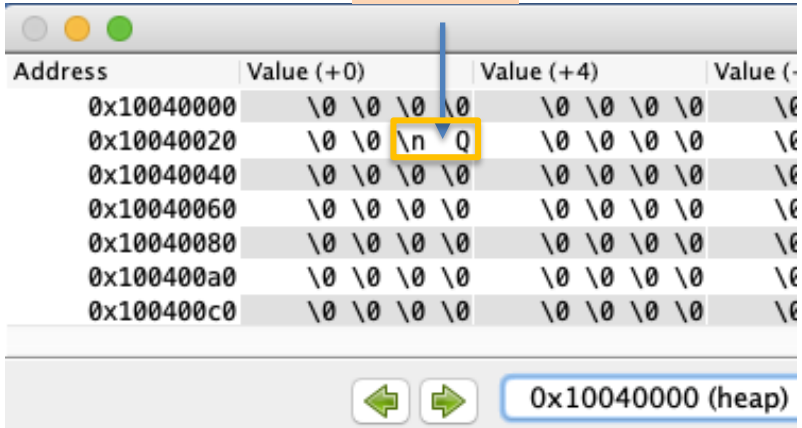

```
#code
.text
La $a0, header
Jal print
Li $t0, stop #limit
loop_main:
    Jal GUI_in #get command
    getChar #$a0= char
    la $t1, filler
    sb $a0, ($t1) #cat char (drop in)
    msgbox Com_str
    Tnei $a0, Qx #Trap: call handler (in ktext)
    #check for infinite loop (#>stop)
    subiu $t0,$t0, 1 #decr stop
    blez $t0, stopped
Bne $a0, Qx, loop_main
#user Quit
La $a0, Quit_msg
Jal print
done #macro for exit#1
stopped:
La $a0, Stop_msg
Jal print
done #macro for exit#2
#---end of main---
```

Recognize "Q"?

.text



In_buf



Address	Value (+0)	Value (+4)	Value (-
0x10040000	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x10040020	\0 \0 \n Q	\0 \0 \0 \0	\0
0x10040040	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x10040060	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x10040080	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x100400a0	\0 \0 \0 \0	\0 \0 \0 \0	\0
0x100400c0	\0 \0 \0 \0	\0 \0 \0 \0	\0

0x10040000 (heap)

"stop" = 3

```
Project 2:  Digital Lab by Jeff D
starting command handler...
starting command handler...
starting command handler...
Stopped out!
-- program is finished running --
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

Proj 1: Handlers in .ktext

.kdata

```
#start handler code in kernel memory
.kdata
kmsg: .asciiz "starting command handler...\n"
Istr: .asciiz "Display initials"
Bstr: .asciiz "Blink"
Fstr: .asciiz "Flash"
Cstr: .asciiz "Calculator"
Xstr: .asciiz "Error: bad command"
.macro push
move $k0, $v0 #save regs
move $k1, $a0
.end_macro
.macro pop_ret
move $v0, $k0 #restore regs
move $a0, $k1
eret
.end_macro
```

New macros

"eret" in macro

.ktext

```

.ktext hdlr_seg
push
print_mac kmsg #prt msg via macro
mfc0 $k0, $14 #EPC
addi $k0,$k0, 4 #incr RA in EPC
mtc0 $k0, $14 #EPC+4 (for ERET)
getChar #re-load: $a0= char
#--Branch Table--
Beq $a0, Ix, I
Beq $a0, Bx, B
```

Complete this

```
Jump target word address beyond 26-bit range
```

Jal print

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x90000000	r a t s	g n i t	m o c	d n a m	n a h	r e l d	\0 . . .	\0 \0 \0 \0
0x90000020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x90000040	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x90000060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x90000080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x900000a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x900000c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x90000000 (.kdata) Hexadecimal Addresses Hexadecimal Values ASCII

Proj 1 Code: Trap Handler

.ktext

```
# Trap handler in the standard MIPS32 kernel text segment

   .ktext 0x80000180
   move $k0,$v0    # Save $v0 value
   move $k1,$a0    # Save $a0 value
   la   $a0, msg   # address of string to print
   li   $v0, 4     # Print String service
   syscall
   move $v0,$k0    # Restore $v0
   move $a0,$k1    # Restore $a0
   mfc0 $k0,$14    # Coprocessor 0 register $14 has address of trapping instruction
   addi $k0,$k0,4  # Add 4 to point to next instruction
   mtc0 $k0,$14    # Store new address back into $14
   eret            # Error return; set PC to value in $14
   .kdata
msg:
   .asciiz "Trap generated"
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Project 1: Digital Lab

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Simulating the Hexa Keyboard and Seven segment display

This tool is composed of 3 parts : two seven-segment displays, an hexadecimal keyboard and counter

Seven segment display

Byte value at address 0xFFFF0010 : command right seven segment display
Byte value at address 0xFFFF0011 : command left seven segment display
Each bit of these two bytes are connected to segments (bit 0 for a segment, 1 for b segment and 7 for point

Hexadecimal keyboard

Byte value at address 0xFFFF0012 : command row number of hexadecimal keyboard (bit 0 to 3) and enable keyboard interrupt (bit 7)
Byte value at address 0xFFFF0014 : receive row and column of the key pressed, 0 if not key pressed
The mips program have to scan, one by one, each row (send 1,2,4,8...) and then observe if a key is pressed (that mean byte value at adresse 0xFFFF0014 is different from zero). This byte value is composed of row number (4 left bits) and column number (4 right bits) Here you'll find the code for each key : 0x11,0x21,0x41,0x81,0x12,0x22,0x42,0x82,0x14,0x24,0x44,0x84,0x18,0x28,0x48,0x88.
For exemple key number 2 return 0x41, that mean the key is on column 3 and row 1.
If keyboard interruption is enable, an exception is started, with cause register bit number 11 set.

Counter

Byte value at address 0xFFFF0013 : If one bit of this byte is set, the counter interruption is enable.
If counter interruption is enable, every 30 instructions, an exception is started with cause register bit number

```
.eqv LEDleft, 0xffff0011
.eqv LEDrt, 0xffff0010
.eqv Jeff, 0x0e
.eqv Drob, 0x5e
.macro push
```

```
la $a1, LEDleft
la $a2, LEDrt
#--Branch Table--
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Proj 1: Memory Segments

.data

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	j o r P	t c e	: 2	i g i D	l a t	b a L	J y b	f f e
0x10010020	\0 \0 \n D	e t n E	o c r	n a m m	: d	/ B / I	/ C / F	\0 \0 \0 Q
0x10010040	w o N	f r e p	i m r o	c g n	a m m o	: d n	* * \0 B	\0 \0 \0 \0
0x10010060	p o t S	d e p	! t u o	e s u \0	u Q r	\n \0 t i	\0 \0 \0	\0 \0 \0 \0
0x10010080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x100100c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0	

.kdata

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x90000000	r a t s	g n i t	m o c	d n a m	n a h	r e l d	\n . . .	s i D \0
0x90000020	y a l p	i n i	l a i t	l B \0 s	\0 k n i	s a l F	a C \0 h	l u c l
0x90000040	r o t a	r r E \0	: r o	d a b	m m o c	\0 d n a	\0 \0 \0 \0	\0 \0 \0 \0
0x90000060	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x90000080	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x900000a0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0
0x900000c0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0	

0x90000000 (.kdata) Hexadecimal Addresses Hexadecimal Values ASCII

MMIO: LED patterns = 0x**0e** + **0x5e**

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0xffff0000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000e5e	0x00000000	0x00000000	0x00000000
0xffff0020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0xffff0040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0xffff0060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0xffff0080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0xffff00a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000		
0xffff00c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000		

0xffff0000 (MMIO) Hexadecimal Addresses Hexadecimal Values

Digita

Project 1

New ***Branch Table***

```
#--Branch Table--
Beq $s0, Ix, I
Beq $s0, Bx, B
Beq $s0, Fx, F
Beq $s0, Zx, Z
Beq $s0, Cx, C
#bad command
_msgbox Xstr
pop_ret #return
#end Br table
```

key ***Subs***

```
#load Initials sub
loadI:
li $t5, Jeff
sb $t5, LEDleft
li $t6, Drob
sb $t6, LEDrt
jr $ra
#delay sub
delay:
subu $t0, $t0, 1
bgtz $t0, delay
jr $ra
###end subs
```

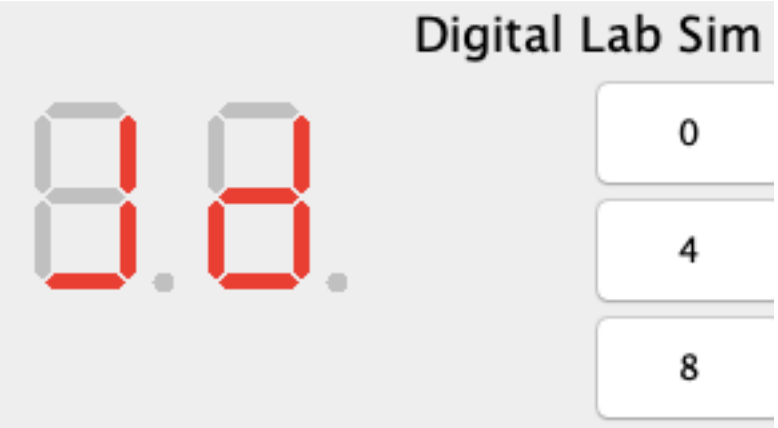
Delay loop

Proj 1: "B" Handler

.ktext

```
141  li $a1, 5 #num blinks-outside loop!
142  B_loop:
143  jal delay
144  sb $zero, LEDleft
145  sb $zero, LEDrt
146  jal delay
147  sb $t5, LEDleft
148  sb $t6, LEDrt
149  subu $a1, $a1, 1
150  bgtz $a1, B_loop
151  pop_ret
152  F:
153  msgbox Fstr
154  pop_ret
155  C:
156  msgbox Cstr
157  pop_ret
158  #delay sub
159  li $a0, 25 #delay (outside loop)
160  delay:
161  subu $a0, $a0, 1
162  bgtz $a0, delay
163  jr $ra
164  #--end of program--#
```

- Key!
- Blink!
- Fiddle with this number (blink rate)

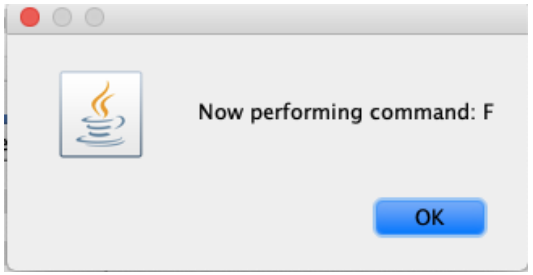



```
Project 2:  Digital Lab by Jeff D
starting command handler...
starting command handler...
starting command handler...
Stopped out!
-- program is finished running --
```

Flash (F)

.ktext

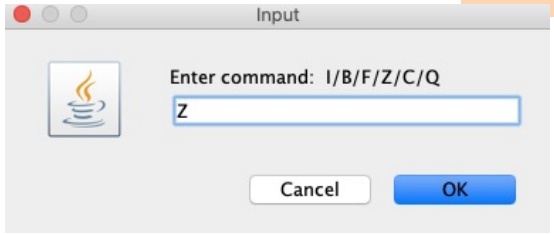
```
F:
jal loadI #reset
_msgbox Fstr
li $s4, 2 #num loops
li $t0, 5 #5/10000 delay
#doubly nested loop
F_outer: #reset patterns/loops
li $s0, 1 # L seg=1
li $s1, 0x80 # R seg=1
li $s5, 8 #num flashes per loop
F_loop:
sb $zero, LEDleft #blank L
sb $zero, LEDrt #blank R
jal delay
rol $s0, $s0, 1 #shift bit L
sb $s0, LEDleft
ror $s1, $s1, 1 #shift bit L
sb $s1, LEDrt
jal delay
   subu $s5, $s5, 1
   bgtz $s5, F_loop
subu $s4, $s4, 1
bgtz $s4, F_outer
pop_ret
#--end Flash
```

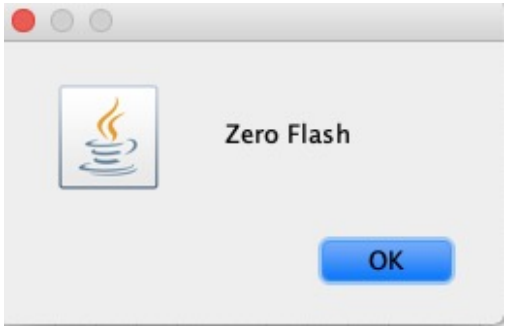


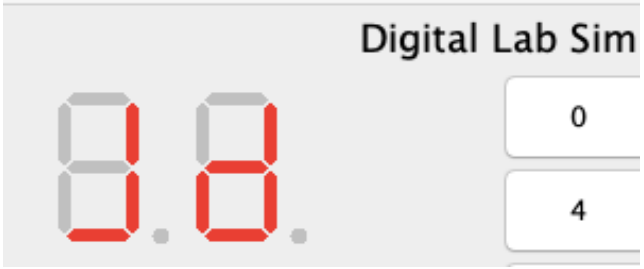
Zero Flash (Z)

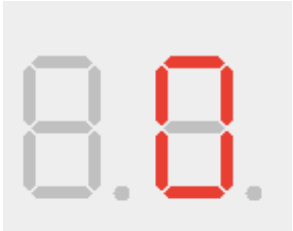
.ktext

```
#Zero Flash
Z:
jal loadI
_msgbox Zstr
li $s5, 10 #num flashes
li $s0, zero
li $s1, 0 #blank
li $t0, 5 #5/10000 delay
Z_loop:
sb $s0, LEDleft
sb $s1, LEDrt
jal delay
#reverse
sb $s1, LEDleft
sb $s0, LEDrt
jal delay
subu $s5,$s5,1
bgtz $s5, Z_loop
pop_ret
#--end Z
```









Lab

C

Data Types in C

❖ Integers (16-bit)

- unsigned int (0 to 65,535)
- signed int (-32,768 to +32,767)

❖ Char string/byte (8-bit)

- unsigned char (0 to 255)
- signed char (-128 to +127)

❖ Floating Point ("Real") (32/64-bit)

- Float (32-bit single precision: +-N x 10^64)
- Double (64-bit double precision: +-N x 10^256)

1.123456789 E+64

Interrupts C Example


```
#include <p18f4321.h>
void    ISR (void);  //declares ISR as sub after main
#pragma code Int=0x08
void  Intasm( )
{
_asm   //use assembly code
GOTO ISR
_endasm
}
#pragma code  //org main
Void main(  )
   {
// do stuff here
While(1) {    }
}
#pragma interrupt ISR
Void ISR (void)     //interrupt svc routine
{   //do int stuff
   }
```

Config Interrupts in C

EXTERNAL INTS | PIC18F

SET ORGS

```
Void   ISR( );  //declare the "ISR" subroutine
#pragma code int_vectH = 0x08  //assign int "vector" for High Pri Int
Void   IntH( )  {
_asm  //use assembly code here (no "GOTO" in C)
GOTO ISR
_endasm
#pragma code int_vectL = 0x18  //assign int "vector" for Low Pri Int
#pragma code  //main starts here (after the Ints)
Void   main ( )
{
ADCON1=0x0F;   //config Port B as input for interrupts
INTCONbits.INT0IE = 1  //enable INT0
INTCON3bits.INT1IE = 1  //enable INT1
INTCONbits.INT0F = 0  //clear flag
INTCON3bits.INT1F = 0 //clear flag
INTCON3bits.INT1IP = 0  //set INT1 to Low priority
RCONbits.IPEN = 1  //enable all priority interrupts
INTCONbits.GIEH = 1  //enable Global High priority interrupts
INTCONbits.GIEL = 1   //enable Global Low priority interrupts
While(1);   //wait for INT0 or INT1
}
```

Or use a "Call": ISR( )

SETUP

PINS

- INT 0 → RB0
- INT 1 → RB1
- INT 2 → RB2

Timers

PIC18F

Timers

- Timer 0
- Timer 1
- Timer 2
- Timer 3
- Watchdog

PRESCALER

- Clock pulse counter (a power of 2)
  - 2
  - 4
  - 8
  - 16
  - 32
  - 64
  - 128
  - 256

MODES

- Timers
  - Count UP or DOWN
  - Preset start value
  - INT on OVERFLOW
  - Use INTERNAL clock
- Counters
  - Use EXTERNAL clock (pin TnCKI)

Config Timers in C

PIC18F

Timers

- Timer 0
- Timer 1
- Timer 2
- Timer 3
- Watchdog

```
#pragma code  //main starts here (after the Ints)
Void   main ( )
{
T0CON=0x06;   //config T0
TMR0H = 0XFF  //init T0 High byte
TMR0L = 0XF0  //init T0 Low byte =-16 (will count UP to 0)
INTCONbits.TMR0IF = 0  //clear Timer0 flag

T0CONbits.TMR0ON = 1  //start timer
While(INTCONbits.TMR0IF ==0);  //wait for INT0 or INT1
T0CONbits.TMR0ON = 0  //stop timer
While(1);    //halt
}
```

SETUP

RUN TIMER

Using Timer0 to create a **1ms delay**, assume:

- 8MHz crystal (=2MHz Instruction rate)
- Prescale value = 128 (2000/128=16)

Lab

Misc Labs

Lab: Temp Conversion

F = C * (9/5) + 32
C = (F-32) * (5/9)

- ❑ (9/5) = 1.8
- ❑ (5/9) = 0.555…

→ how to represent a fixed point number?

→ Consider using BCD:

- MULT by 18 or 556
- divide by 10 or 1000 by shifting BCD digits right