Computer Organization & Design

External Calls

(System Calls)

External Calls

- Functions provided to an executing program by system level software (operating system)
- Most system calls involve I/O
  - Input from user via keyboard
  - Output to console
  - File I/O
  - Other utilities

External Call Codes

- An integer value indicating the type of request
- **a7** is the designated register for the external call code in RISC-V

1	print integer
2	print single float
4	print string
5	read integer
6	read single float
8	read string
9	system break
10	program exit
11	print ASCII character
12	read character
34	print integer (hex)
35	print integer (binary)
36	print integer (unsigned)
57	close a file
63	read from a file
64	write to a file
1024	open a file

- There are additional external calls defined but we seldom use them in programs for this class

External Call Parameters

- Most external calls require that you furnish parameters or arguments
  - If you want to print an integer value, you must provide the value to be printed in register **a0**
  - If you want to print a string, you must provide the memory address where the string is defined in register **a0**
  - If you want to print a character, you must provide the ASCII code for that character in register **a0**
- **a0 – a6** function as argument registers for external calls (integer, character, string)
- **fa0** is used for floating-point values read or printed (when we get there in Chapter 3)

Return Values

- Some external calls return a value to the program
  - E.g. user inputs a number, character or string
- **a0** is used for integer and character return values
  - When a character is entered, no carriage return is needed; the character is read as soon as it is typed
  - This is called a “single character read”
- **fa0** is used for floating-point return values

Strings as Input

- Strings are written directly to memory at the address specified in the **a0** register prior to the external call
  - Length of string is determined by the value in register **a1** (***includes the null character***)
    - If a1 = 1, only a null character is written
    - If a1 < 1, nothing is written
  - When the number of characters specified by **a1** is reached (a1 minus 1), input automatically terminates
  - A carriage return will terminate the string but the newline character is added to the string value and becomes part of the string
  - The null character is added to the end of the string

External Call Process

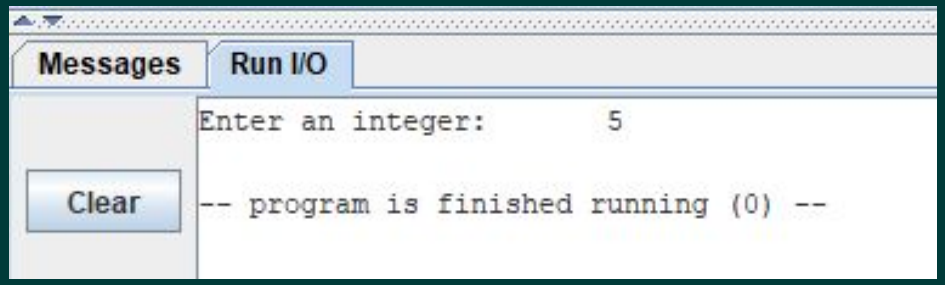
- Place the external call code value in **a7**
- Place required argument values in **a0, a1** …
- Execute ecall instruction
- Copy any return values from **a0** or **fa0** to other registers
  - Subsequent external calls will overwrite the return registers

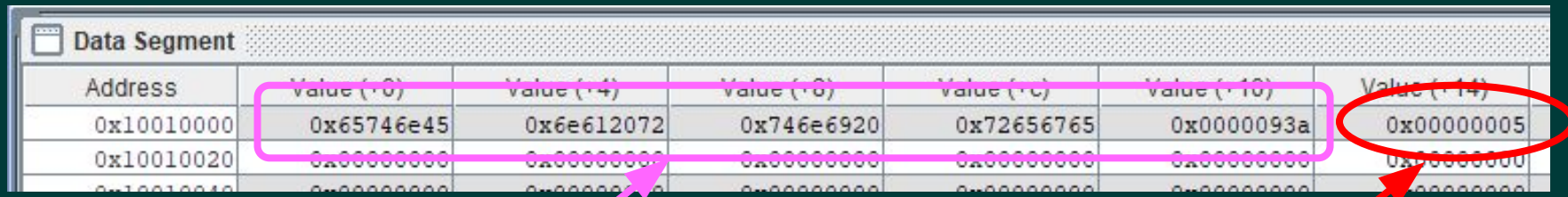
Sample Program

```
        .data
prompt: .asciz  “Enter an integer:\t“
reply:    .word    0

        .text
main:    lui   s0, 0x10010      # memory base address in s0
        ori   a7, zero, 4   # external code for print string in a7
        or    a0, zero, s0 # load address of string in a0
        ecall                 # print string on console

        ori   a7, zero, 5   # external code for reading an integer
        ecall                 # read reply - value placed in a0
        sw   a0, 20(s0)    # store the read number into memory

exit:       ori   a7, zero, 10 # external call code for ending program
        ecall
```

Note: control characters (tab, newline) can be included in a string definition.

Output of Program





Prompt:

Entered value stored in memory.

Console Output Formatting

- All aspects of console output must be coded by the assembly language programmer.
  - If you want to separate values on a single line by a space or tab, you have print these values to the console at the appropriate locations
  - If you want to begin output on a new line, then you have to print the new line character to the console.
  - All of these output format characters are part of the ASCII character set and can be printed by defining them as arguments to a system call
  - Shortcut: if you want to tab or go to a new line after printing a string, you can include \t or \n to the end of the string definition

Sample Program with Formatting Ecalls

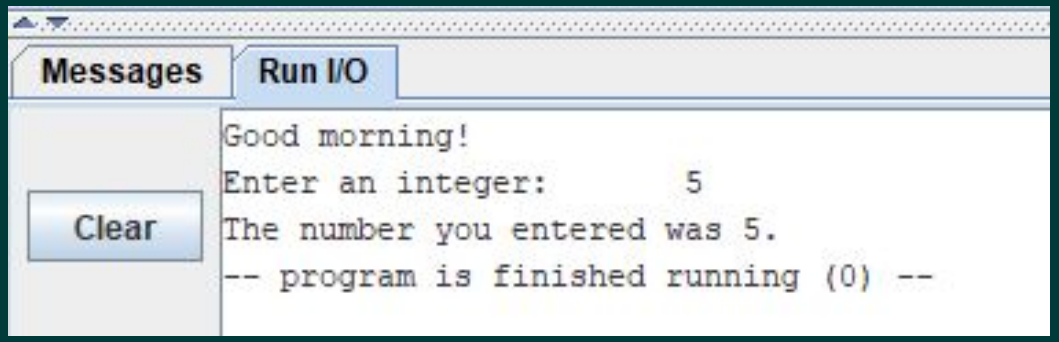
```
        .data
outstr1:    .asciz      "Good morning!\n"
outstr2:    .asciz      "Enter an integer:\t"
outstr3:    .asciz      "The number you entered was "

        .text
main:       lui  s0, 0x10010  # define memory base addr

        ori  a7, zero, 4     # external code print string
        or   a0, zero, s0    # address of outstr1
        ecall                # print string
        addi    a0, s0, 15# address of outstr2
        ecall                # print string
```

Sample Program (continued)

```
    ori a7, zero, 5         # external code read integer
    ecall                   # read integer
    or  t0, zero, a0        # copy value to t0
    ori a7, zero, 4         # external code print string
    addi    a0, s0, 34      # address of outstr3
    ecall                   # print string
    ori a7, zero, 1         # external code print integer
    or  a0, zero, t0        # copy value to a0
    ecall                   # print value
    ori a7, zero, 11        # external code print character
    ori a0, zero, 0x2e      # ASCII code for period
    ecall                   # print period
exit:   ori a7, zero, 10        # external code for exit
    ecall                   # exit program
```

Console Output of Sample Program

Good morning!
Enter an integer: 5
The number you entered was 5.
-- program is finished running (0) --



Simplifying Main Program

- Console I/O requires multiple instructions
- The more I/O the longer an assembly language program will become
- Many of the functions may be identical but with different values
- Using subroutines for I/O is an appropriate way to reduce the size of a main program making it easier to understand and debug
- Subroutines are the next topic

Computer Organization & Design

Subroutines

The assembly language version of procedures, functions and methods

Procedures, Functions, Methods

- All HLLs provide for subdividing work into smaller parts
- Structured programming uses procedures and functions
- Object-oriented programming uses methods and classes
- There is a direct correlation between HLL techniques of subdivision and assembly (machine) language constructs

Subroutines

- Subroutines are the procedures, functions, methods of assembly language
  - Makes complex programs easier to understand
  - Allows reuse of code
  - Allows programmers to focus on a portion of a task
  - Helps reduce the length of main program

- Must adhere to specific programming rules
  - Appropriate register use
  - Global vs local data & how to define at assembly level
  - Storage allocation
  - Returning from the subroutine

Normal Steps in Subroutine Execution

- Place data needed by the subroutine (parameters, arguments) where they can be accessed
- Transfer control to the subroutine
- Allocate storage resources in the subroutine (registers, stack) required by the subroutine
- Execute the subroutine
- Place results data in a place where the calling program can access them
- Return control to the point of origin

Subroutine Register Usage

- In RISC-V, the following registers are used for subroutine calls
  - **a0 … a7** argument registers to pass data and and for return values from subroutines
  - **ra** return address register
  - **s0 … s7** callee saved registers – must be preserved and restored by callee

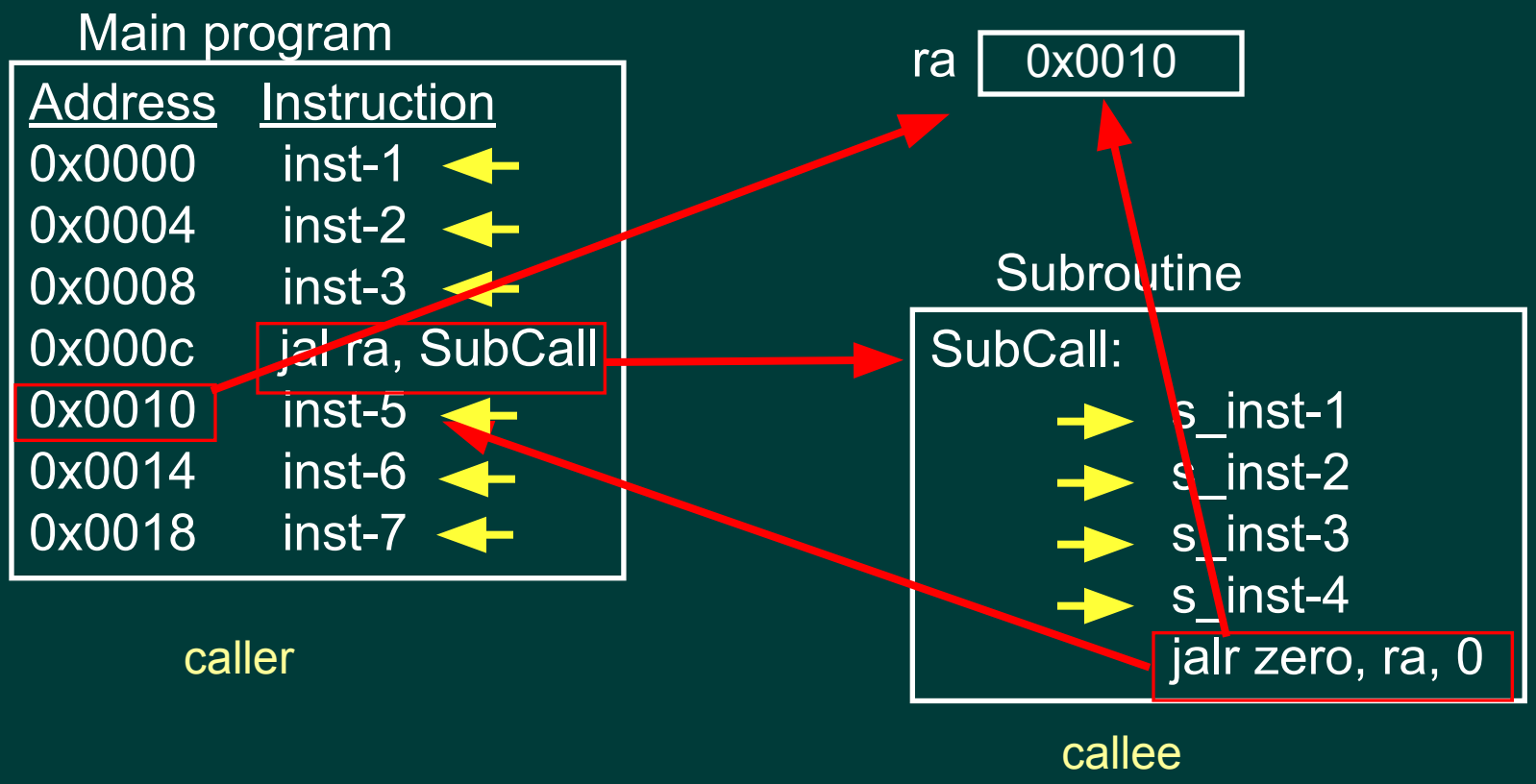
Caller & Callee Terminology

- The **caller** transfers execution to a subroutine
  - We say the subroutine was called
  - The caller is usually the main program but subroutines themselves can call other subroutines
- The **callee** is the subroutine that is to be executed when called
  - Subroutines will return back to the caller at the point the subroutine was called

Subroutine Instructions

- Jump and link (call the subroutine)
  - Format: jal ra, SubroutineAddress
  - Jumps to address of the subroutine and automatically saves the address of the next instruction to be returned to in register **ra**
- Jump and link register (return from subroutine)
  - Format: jalr zero, ra, 0
  - Last instruction in the subroutine and is the link back to the calling program

How Subroutine Calls Work

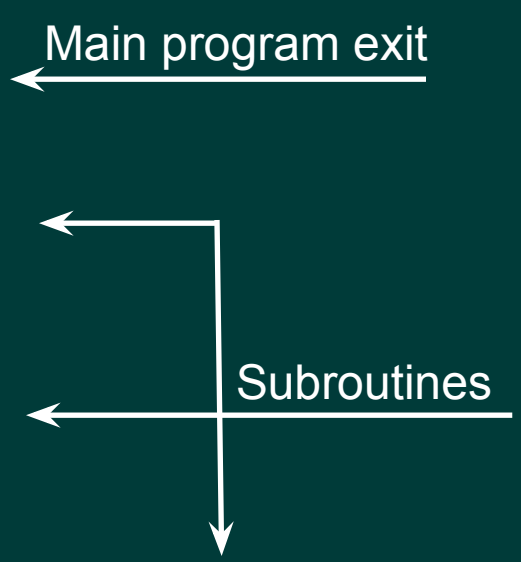


Since the program counter was incremented during the fetch of the jal instruction, it is the value of the PC that is saved into the ra register.

Where Do Subroutines Go?

- At the end of the assembly language program
- After the exit code instructions
- Subroutines are still part of the .text segment

```
        .data

        .text
main:   jal     sub1
        jal     sub2

exit:   ori     a7, zero, 10
        ecall

sub1:

        jalr    zero, ra, 0

sub2:

        jalr    zero, ra, 0
```



Simple Subroutine Call Program

```
        .data
num1:   .word 0

        .text
main:   lui   s0, 0x10010
        jal   ra, sub1          # call sub1
        jal   ra, sub2          # call sub2
exit:   ori   a7, zero, 10
        ecall

sub1:   sw    s0, 0(s0)         # store s0
        jalr  zero, ra, 0       # return

sub2:   lw    s1, 0(s0)         # load s1
        jalr  zero, ra, 0       # return
```

Global vs. Local Data

- HLL construct that differentiates data accessible from all code segments vs. data accessible only within the code block where it is defined
- Defining the concept of scope at the assembly language level is completely up to the programmer
  - Generally, the use of the stack, thread and global pointer registers can be used to further segment the data segment for local vs. global data
- Multiple data segments can also be defined within programs specifically for data to be accessed within subroutines

Stack

- LIFO (last-in first-out) data structure
- Where is it? In memory – more specifically, in high program data space.
- Push – place an item on the top of the stack
- Pop – take an item off the top of the stack
- Stack grows from higher addresses to lower addresses
- In RISC-V assembler, use stack pointer register (**sp**)
  - Pushing values requires subtracting from sp
  - Popping values requires adding to sp

Stacks (continued)

- No push or pop assembly language instructions
- Programmer must manipulate the stack in code
- Two methods:
  - Allocating fixed number of locations in stack for data
    - sp is modified first, then data are stored using appropriate offsets
  - Dynamically manipulating stack
    - Data are stored first and sp is modified after each push operation
    - May not provide best performance but used when amount of data to be pushed is indeterminate

Code Example (Fixed Stack)

- First instructions in subroutine allocate and push data onto the stack

```
addi    sp, sp, -12     # add immediate
sw s3, 0(sp)
sw s4, 4(sp)
sw s5, 8(sp)
```

- Must know how many items you need to push and data size
  - sp value adjusted in one instruction
  - Pushes done with sw instructions using constant offset values
  - Alignment restriction still applies

Code Example (continued)

- Last instructions in subroutine restore s registers and reset value of sp

```
lw  s3, 0(sp)
lw  s4, 4(sp)
lw  s5, 8(sp)
addi   sp, sp, 12
```

- Note the restore (pop) is reverse order of push
- Only the needed s registers are saved and restored
  - Recall s (save) registers contain data that need to be retained across subroutine calls
  - Standard protocol for s usage

Same Example (Dynamic Allocation)

- To push, store data item, then immediately decrement sp
- To pop, increment sp, then load data item

Push sequence

```
sw s3, 0(sp)
addi    sp, sp, -4
sw s4, 0(sp)
addi    sp, sp, -4
sw s5, 0(sp)
addi    sp, sp, -4
```

Pop sequence

```
addi    sp, sp 4
lw  s5, 0(sp)
addi    sp, sp 4
lw  s4, 0(sp)
addi    sp, sp 4
lw  s3, 0(sp)
```

- Method used when number of items is not known or stack access is done within a loop

Nested Subroutines

- Subroutines that call other subroutines
  - Also includes recursive subroutine calls
- What do we do with only one stack pointer?
- We use additional registers
  - Thread pointer (tp) – subroutine base address
    - Register 8 (can also be used as a frame pointer (fp)
  - Global pointer (gp) – static data base address
    - Register 3
- Caller and callee take responsibility for saving required data
  - Caller pushes required data onto stack before subroutine call
  - Callee pushes conventional data onto stack

Nested Subroutines (continued)

- Frame pointer contains address for the beginning of data saved by the caller
  - Saved argument registers (a registers)
  - Saved return address (ra register)
  - Saved local data (s and t registers)
- Stack pointer used by callee for s registers
- Global pointer contains base address of any globally declared data used by all subroutines

Example Program

- Defines multiple data segments for global vs. local data

```
        .data         # global data segment
num1:   .byte 10

        .text
main:   lui   s0, 0x10010
        jal   ra, sub1
exit:   ori   a7, zero, 10
        ecall

        .data 0x10010064          # local data for subroutine
num2:   .byte 20

        .text              # continuation of text segment
sub1:   sw    s0, 0(sp)          # save global memory base address
        addi  sp, sp, -4         # decrement stack pointer
        addi  s0, s0, 0x64       # define local memory base address
        lw    t0, 0(s0)          # access local data
        addi  sp, sp 4           # increment stack pointer
        lw    s0, 0(sp)          # restore global memory base address
        jalr  zero, ra, 0        # return from subroutine
```

Computer Organization & Design

Pseudoinstructions

(sometimes called macro instructions)

Caution Required!!

Pseudoinstructions

- Pseudoinstructions are often common variations of machine language instructions
- They are not basic instructions and are not directly executed by the hardware
- Pseudoinstructions are used to extend the instruction set
- Sometimes makes assembly language programming a little easier
- Pseudoinstruction use in RARS requires changing the setting to allow them
- Be careful to follow syntax rules for basic instructions

Pseudoinstructions

- The assembler translates pseudoinstructions into basic (hardware native) instructions when the machine code is generated
- Each pseudoinstruction may represent one or more basic instructions
- Pseudoinstructions should not be used in place of basic instructions unless it adds clarity to the program

Use pseudoinstructions for assignments in this class only if directed to do so and use only those explicitly stated!

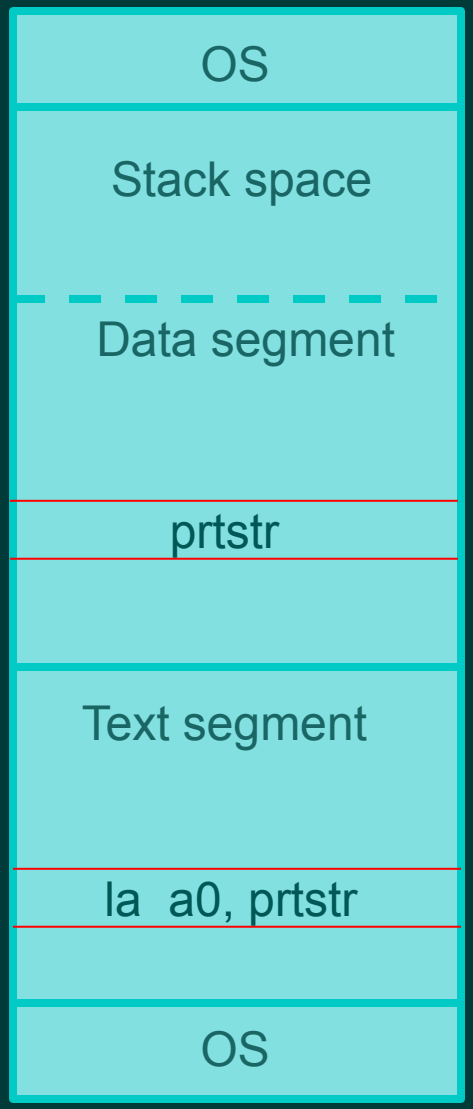
Example Pseudoinstruction

- Load Address (la)
  - This pseudoinstruction loads the address of a value into a register
  - Requires only knowledge of symbol name
  - Commonly used for string values associated with I/O
  - Syntax: la reg, symbol_name
  - Example: la a0, string
    - This instruction places the memory address of the symbol string into register a0
  - Conversion: auipc # add upper immediate to PC
    addi # add immediate offset
  - Assembler uses internal symbol table to derive 20-bit immediate and 12-bit immediate offset

AUIPC Instruction

- Add upper immediate to PC
- This instruction adds a 20-bit immediate value to the upper 20 bits of the program counter
- This instruction enables PC-relative addressing in RISC-V
- To form a complete 32-bit PC-relative address, auipc forms a partial result
- A subsequent addi instruction adds in the lower 12 bits to complete the 32-bit address

Load Address Conversion



Assembler calculates the distance from the current instruction to the data value.

This value is parsed into the upper 20 bits and the lower 12 bits.

The auipc instruction is generated.

The addi instruction is generated to add 10.

0x10010022

0x10010000

0x00400018

0x00400000

```
auipc   a0, 0x0fc10
addi    a0, a0, 0x10
```

0x0fc1000a

lower 12 bits

upper 20 bits

Other Pseudoinstruction Examples

- mv copy value from one register to another
  - Converted to add instruction with zero register
    - Example: mv t1, t0 add t1, zero, t0

- neg calculate the 2's complement value
  - Converted to subtract instruction with zero register
    - Example: neg t1, t0 sub t1, zero, t0

Other Pseudoinstruction Examples

- nop no operation
  - Converted to add immediate with zero & constant 0
    - Example: nop addi zero, zero, 0

- not bitwise complement of a value
  - Converted to exclusive or immediate operation with -1 constant
    - Example: not t1, t0 xori t1, t0, -1