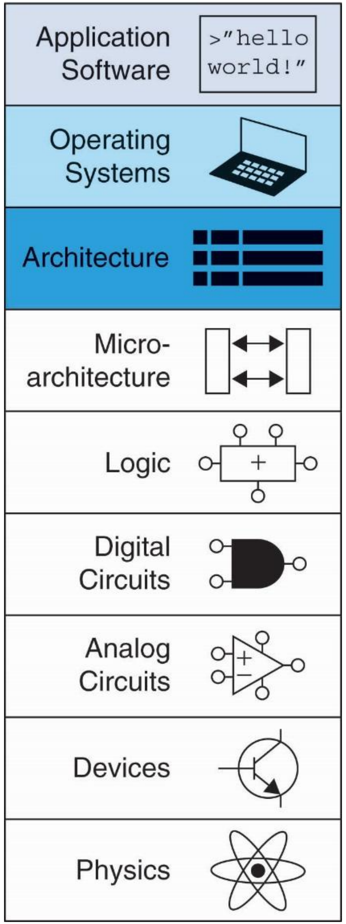
COSC175 (Systems I): Computer Organization & Design

Professor Lillian Pentecost
Fall 2024

Warm-Up November 5

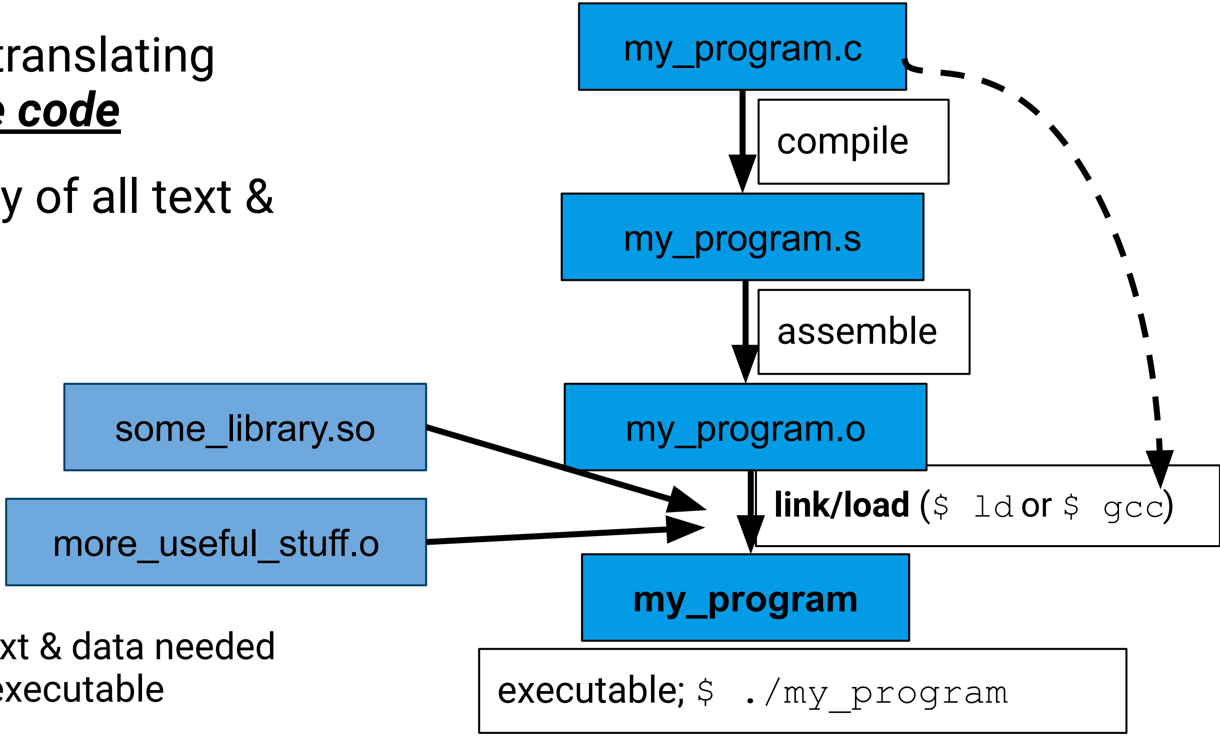
- Where we were
  - How is memory organized, and how is data managed across function calls?
- Where we are going
  - How are high-level languages translated into assembly instructions?
  - Picking back up with memory allocation and function calls, from a C perspective and a RISC-V assembly perspective
- Logistics, Reminders
  - TA help 7-9PM on Sundays, Tuesdays, Thursdays in C107
  - LP Office hours M 9-10:30AM, Th 2:30-4PM
  - Weekly Exercises due Friday 5PM
  - **15-minute Pre-Lab for tomorrow**



How do we generate executable programs?

Each command is a step to translating down to ***executable machine code***

- The **executable** is a copy of all text & static data



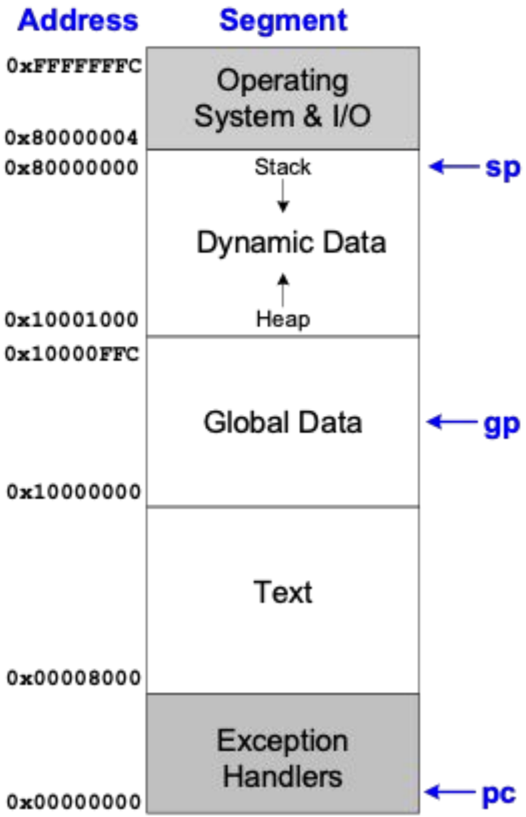
- **Linker-loader:**
  - **Links** / Appends other text & data needed
  - **Loads** / Generates final executable

RISC-V Register Set

Name	Register Number	Usage
`zero`	x0	Constant value 0
`ra`	x1	Return address
`sp`	x2	Stack pointer
`gp`	x3	Global pointer
`tp`	x4	Thread pointer
`t0-2`	x5-7	Temporaries
`s0/fp`	x8	Saved register / Frame pointer
`s1`	x9	Saved register
`a0-1`	x10-11	Function arguments / return values
`a2-7`	x12-17	Function arguments
`s2-11`	x18-27	Saved registers
`t3-6`	x28-31	Temporaries

Example RISC-V Memory Map

- **Instructions** (also called *text*)
- **Data**
  - **Global/static:** allocated before program begins
  - **Dynamic:** allocated within program
- Special registers to track important addresses:
  - **pc: tracks the memory address of the current instruction**
  - sp: address of top of the stack



A first C program

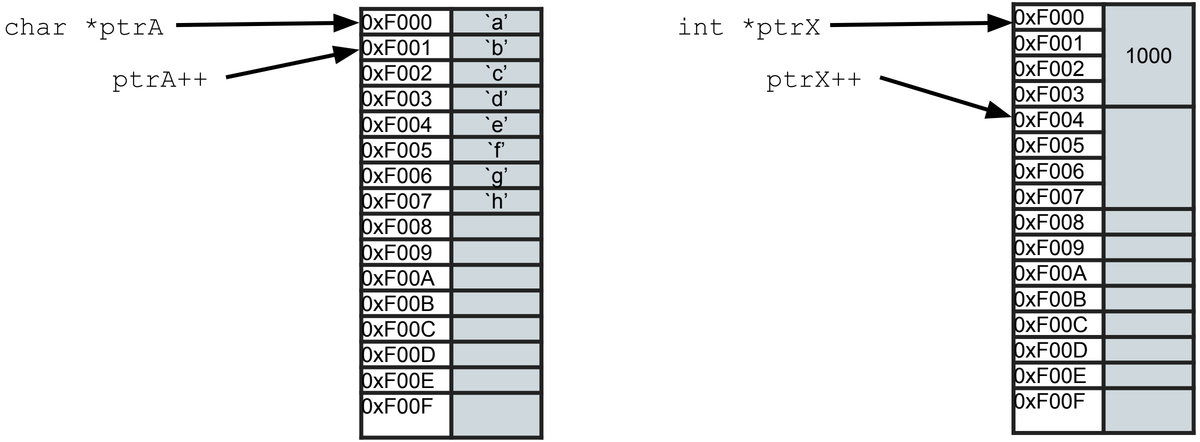
- Variable Declarations
- Printing to Screen
- Variable Types
    - Ones you are familiar with (`char`, `int`, `double`, `float`)
    - At least one important one you may not be: ***pointers***
        - Store an address of a value in memory
        - Look like this:
            - `int*` is a pointer to an integer value (32b)
            - `char*` is a pointer to a character value (8b)

Pointers I

- Store an address of a value in memory
- Several ways to initialize
  - A safe way to start: set to `NULL`
    - `int *ptrA = NULL;`
  - Alternatively: set using address of existing value, using `&` to get address of variable
    - `int a = 100;`
    - `int *ptrA = &a;`
- How to access the ***<u>value</u>***? Dereferencing, like in a **sw** or **lw** in assembly!
  - Given a pointer to an `int`:
    - `int a = *ptrA; //deference ptrA using * to retrieve, assign value`
    - `*ptrA = 100;`
  - You can read or write a value to the address at ptrA via dereferencing
  - What happens if you dereference `NULL`?

Pointers II

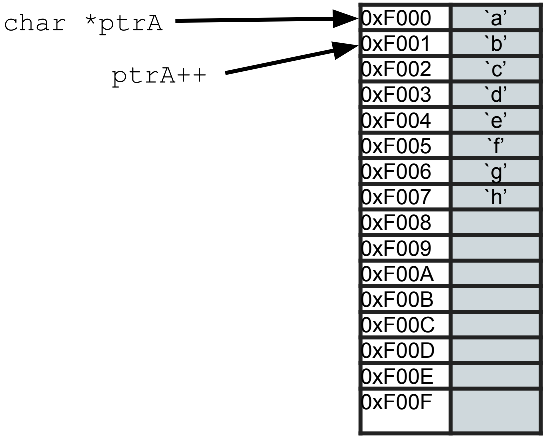
- Pointers must have a ***type*** too – a pointer of type `int` references a 4-byte block, while a pointer to type `char` references a 1 byte block

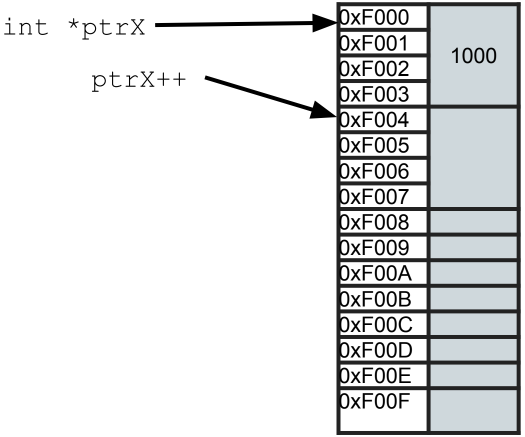


- IMPORTANT: pointers can be ***aliased*** ; multiple pointer variables may point to the ***same address***
    - ***Why does this matter? How can this be used?***

Pointers III

- Add and subtract from an address to get a new address (when needed)
  - Result depends on the pointer type





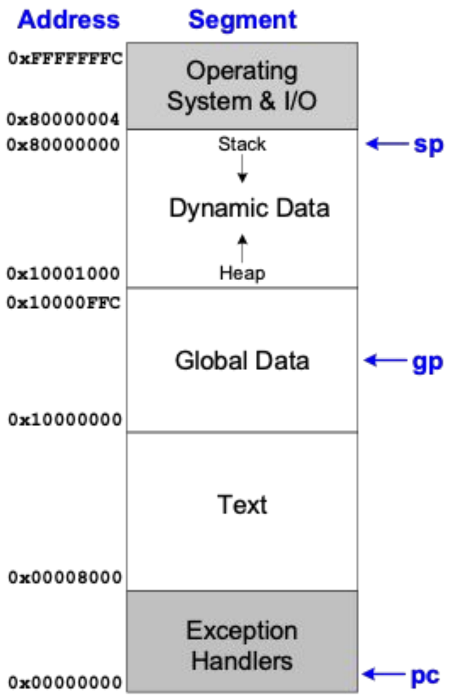
- IMPORTANT: explicitly cast pointer to avoid confusion
  - Example: increment `i` characters away from `ptrA`
    - `char *ptrG = (char*) ptrA + i;`
    - `char *ptrG = (char*) ptrA + sizeof(char) * i;`
  - For `i=6` and `ptrA=0xF000`, this would evaluate to `ptrG = 0xF000 + 1 * 6 = 0xF006`

Array variable declarations

Your program can allocate memory on the `stack` or on the `heap;` Global variables, other things that persist will typically be in `data`

An important mental model for the `stack` vs. `heap`:

- ***Stack*** grows incrementally and continguously; typically has a limited overall capacity
- ***Heap*** is an open memory address range that we can claim chunks of for use in our programs (can be a mess!)
    - Heap will be the focus of next class discussion
- ***In either case***, you can use index notation (e.g., A[0] or B[0]) to dereference and retrieve array value at index 0




```
1.  int A[10]; // allocates 10 int on stack, "A" is base address
2.  int* B = malloc(10 * sizeof(int)); // allocates 1 pointer on the stack,
    with value referencing block of 10 int on heap
```

Function Call Summary

- **Caller**
  - Save any needed registers that callee won't preserve
    - (`ra`, maybe `t0-t6/a0-a7`)
  - Put arguments in `a0-a7`
  - Call function: `jal callee`
  - Look for result in `a0`
  - Restore any saved registers
- **Callee**
  - Save registers that might be disturbed `(s0-s11)`
  - Perform function
  - Put result in `a0`
  - Restore registers
  - Return: `jr ra`

Preserved Registers

- What is the callee ***responsible*** for preserving / protecting the value of? AKA ***calling conventions***

Preserved *Callee-Saved*	**Nonpreserved** *Caller-Saved*
`s0-s11`	**`t0-t6`**
`sp`	**`a0-a7`**
`ra`	
stack above **`sp`**	stack below **`sp`**

Storing **Saved Registers** on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -4          # make space on stack to
                           # store one register
  sw   s3, 0(sp)           # save s3 on stack
  add  t0, a0, a1          # t0 = f + g
  add  t1, a2, a3          # t1 = h + i
  sub  s3, t0, t1          # result = (f + g) - (h + i)
  add  a0, s3, zero        # put return value in a0
  lw   s3, 0(sp)           # restore $s3 from stack
  addi sp, sp, 4           # deallocate stack space
  jr   ra                  # return to caller
```

Storing **Saved Registers** on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -4      # make space on stack to
                       # store one register
  sw   s3, 0(sp)       # save s3 on stack
  add  t0, a0, a1      # t0 = f + g
  add  t1, a2, a3      # t1 = h + i
  sub  s3, t0, t1      # result = (f + g) - (h + i)
  add  a0, s3, zero    # put return value in a0
  lw   s3, 0(sp)       # restore $s3 from stack
  addi sp, sp, 4       # deallocate stack space
  jr   ra              # return to caller
```

Storing Saved Registers on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -4         # make space on stack to
                          # store one register
  sw   s3, 0(sp)          # save s3 on stack
  add  t0, a0, a1         # t0 = f + g
  add  t1, a2, a3         # t1 = h + i
  sub  s3, t0, t1         # result = (f + g) - (h + i)
  add  a0, s3, zero       # put return value in a0
  lw   s3, 0(sp)          # restore $s3 from stack
  addi sp, sp, 4          # deallocate stack space
  jr   ra                 # return to caller
```

Storing **Saved Registers** on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -4        # make space on stack to
                         # store one register
  sw   s3, 0(sp)         # save s3 on stack
  add  t0, a0, a1        # t0 = f + g
  add  t1, a2, a3        # t1 = h + i
  sub  s3, t0, t1        # result = (f + g) - (h + i)
  add  a0, s3, zero      # put return value in a0
  lw   s3, 0(sp)         # restore $s3 from stack
  addi sp, sp, 4         # deallocate stack space
  jr   ra                # return to caller
```

Storing **Saved Registers** on the Stack

```
# s3 = result
diffofsums:
   addi sp, sp, -4       # make space on stack to
                         # store one register
   sw   s3, 0(sp)        # save s3 on stack
   add  t0, a0, a1       # t0 = f + g
   add  t1, a2, a3       # t1 = h + i
   sub  s3, t0, t1       # result = (f + g) - (h + i)
   add  a0, s3, zero     # put return value in a0
   lw   s3, 0(sp)        # restore $s3 from stack
   addi sp, sp, 4        # deallocate stack space
   jr   ra               # return to caller
```

Storing **Saved Registers** on the Stack

```
# s3 = result
diffofsums:
   addi sp, sp, -4      # make space on stack to
                        # store one register
   sw   s3, 0(sp)       # save s3 on stack
   add  t0, a0, a1      # t0 = f + g
   add  t1, a2, a3      # t1 = h + i
   sub  s3, t0, t1      # result = (f + g) - (h + i)
   add  a0, s3, zero    # put return value in a0
   lw   s3, 0(sp)       # restore $s3 from stack
   addi sp, sp, 4       # deallocate stack space
   jr   ra              # return to caller
```

Storing **Saved Registers** on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -4      # make space on stack to
                       # store one register
  sw   s3, 0(sp)       # save s3 on stack
  add  t0, a0, a1      # t0 = f + g
  add  t1, a2, a3      # t1 = h + i
  sub  s3, t0, t1      # result = (f + g) - (h + i)
  add  a0, s3, zero    # put return value in a0
  lw   s3, 0(sp)       # restore $s3 from stack
  addi sp, sp, 4       # deallocate stack space
  jr   ra              # return to caller
```

Storing **Saved Registers** on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -4         # make space on stack to
                          # store one register
  sw   s3, 0(sp)          # save s3 on stack
  add  t0, a0, a1         # t0 = f + g
  add  t1, a2, a3         # t1 = h + i
  sub  s3, t0, t1         # result = (f + g) - (h + i)
  add  a0, s3, zero       # put return value in a0
  lw   s3, 0(sp)          # restore $s3 from stack
  addi sp, sp, 4          # deallocate stack space
  jr   ra                 # return to caller
```

Storing **Saved Registers** on the Stack

```
# s3 = result
diffofsums:
   addi sp, sp, -4       # make space on stack to
                         # store one register
   sw   s3, 0(sp)        # save s3 on stack
   add  t0, a0, a1       # t0 = f + g
   add  t1, a2, a3       # t1 = h + i
   sub  s3, t0, t1       # result = (f + g) - (h + i)
   add  a0, s3, zero     # put return value in a0
   lw   s3, 0(sp)        # restore $s3 from stack
   addi sp, sp, 4        # deallocate stack space
   jr   ra               # return to caller
```

Optimized `diffofsums`

```
# what if we just avoided using a saved register, tho?

# a0 = result
diffofsums:
  add  t0, a0, a1    # t0 = f + g
  add  t1, a2, a3    # t1 = h + i
  sub  a0, t0, t1    # result = (f + g) - (h + i)
  jr   ra            # return to caller
```

Non-Leaf Function Calls

Non-leaf function:
a function that calls another function

```
func1:
  addi sp, sp, -4    # make space on stack
  sw   ra, 0(sp)     # save ra on stack
  jal  func2
  ...
  lw   ra, 0(sp)     # restore ra from stack
  addi sp, sp, 4     # deallocate stack space
  jr   ra            # return to caller
```

Must preserve `ra` before function call.

Pseudoinstructions
(for your reference)

- **Pseudoinstructions** are not actual RISC-V instructions but they are often more convenient for the programmer.
- **Assembler** converts them to real RISC-V instructions.

Jump Pseudoinstructions

- RISC-V has four jump psuedoinstructions
  - `j imm` `jal x0, imm`
  - `jal imm` `jal ra, imm`
  - `jr rs` `jalr x0, rs, 0`
  - `ret` `jr ra` (i.e., `jalr x0, ra, 0`)

Labels

- Label indicates where to jump
- Represented in jump as immediate offset
  - **imm** = # bytes past jump instruction
  - In example, below, **imm** = (51C-300) = 0x21C
  - `jal simple = jal ra, 0x21C`

RISC-V assembly code

```
0x00000300 main:    jal   simple        # call
0x00000304          add   s0, s1, s1
...                 ...
```

Long Jumps

- The immediate is limited in size
  - 20 bits for `jal`, 12 bits for `jalr`
  - Limits how far a program can jump
- Special instruction to help jumping further
  - `auipc rd, imm`: add upper immediate to PC
    - $\texttt{rd = PC + \{imm}_{31:12}\texttt{, 12'b0\}}$
- Pseudoinstruction: $\texttt{call imm}_{31:0}$
  - Behaves like `jal imm`, but allows 32-bit immediate offset

$\texttt{auipc ra, imm}_{31:12}$
$\texttt{jalr ra, ra, imm}_{11:0}$

More RISC-V Pseudoinstructions

Pseudoinstruction	RISC-V Instructions
`j label`	`jal zero, label`
`jr ra`	`jalr zero, ra, 0`
`mv t5, s3`	`addi t5, s3, 0`
`not s7, t2`	`xori s7, t2, -1`
`nop`	`addi zero, zero, 0`
`li s8, 0x56789DEF`	`lui s8, 0x5678A` `addi s8, s8, 0xDEF`
`bgt s1, t3, L3`	`blt t3, s1, L3`
`bgez t2, L7`	`bge t2, zero, L7`
`call L1`	auipc ra, $imm_{31:12}$ jalr ra, ra, $imm_{11:0}$
`ret`	`jalr zero, ra, 0`

Wrap-Up November 5

SCAN ME

- Coming up next!
  - Practice with the stack and the heap, allocating memory and building up to “real” programs in C syntax, studying the corresponding assembly
- Logistics, Reminders
  - TA help 7-9PM on Sundays, Tuesdays, Thursdays in C107
  - LP Office hours M 9-10:30AM, Th 2:30-4PM
  - Weekly Exercises due Friday 5PM
  - **15-minute Pre-Lab for tomorrow**
  - Grading Announcement!
    - Through Lab 4, plus midterm feedback available via Moodle
- FEEDBACK
  - https://forms.gle/5Aafcm3iJthX78jx6