RISC-V PROGRAMMING

ABI: Application Binary Interface

- A set of rules that defines how functions pass arguments, return values, and use registers and memory.
- Register use conventions are a part of this!

Register use conventions

By convention, the RISC-V registers are assigned to specific uses and names used in the ABI. These are supported by the assembler, and high-level languages.

We'll use these names increasingly.

Why have such conventions?

x0/zero (always zero)
x1/ra (return address)
x2/sp (stack pointer)
x3/gp (global pointer)
x4/tp (thread pointer)
x5/t0 (temporary)
x6/t1 (temporary)
x7/t2 (temporary)
x8/fp (frame pointer)
x9/s1 (saved)
x10/a0 (argument/return value 1)
x11/a1 (argument/return value 2)
x12/a2 (argument)
x13/a3 (argument)
x14/a4 (argument)
x15/a5 (argument)

x16/a6 (argument)
x17/a7 (argument)
x18/s2 (saved)
x19/s3 (saved)
x20/s4 (saved)
x21/s5 (saved)
x22 (saved)
x23 (saved)
x24 (saved)
x25 (saved)
x26 (saved)
x27 (saved)
x28 (temporary)
x29 (temporary)
X30 (temporary)
X31 (temporary)

Prevents functions from accidentally overwriting each other's values

Allows independently compiled code to co-execute correctly!

Separately written fns can safely share CPU without corrupting each other's data

Assembler Directives

There are also many assembler directives that provide hints to the assembler for allocating memory locations

Directive	Meaning
`.word v1,v2,v3, ...,vn`	Initialize sequential memory words with values v1, v2, etc
`.space N`	Allocate space for N words
`.string "anysizeoftext"`	Initialize sequential memory bytes based on the given string
`.text`	Place the following in the .text segment (usually instructions)
`.data`	Place the following in the .data segment (usually global or static data declarations)
`.align N`	N must be a power of 2. Adjusts the next address in the current segment such that address % N == 0

Loops

- Let’s implement the following code snippet in RISC-V

```
sum = 0;
for (i = 0; i < 5; i++){
  sum += 2;
}
sum += 10;
```

Loops

- We'll start off by initializing **`sum`** and **`i`**. We will also use a register to store the termination point of the loop.
- You can use any register for these values (except x0). Let's use:
  - *x10 = sum*
  - *x8 = i*
  - *x9 = terminating point*

```
# sum = 0, i = 0, limit = 5
addi x10, x0, 0    # sum
addi x8, x0, 0     # i
addi x9, x0, 5     # terminating point
```

Loops

- Let's write the loop body next
- Inside of the loop body, we need to perform the following operations
  - *sum += 2*
  - *i++*

```
# sum = 0, i = 0, limit = 5
addi x10, x0, 0    # sum
addi x8, x0, 0    # i
addi x9, x0, 5    # terminating point
```

```
addi x10, x10, 2 #sum +=2
addi x8, x8, 1 # i++
```

Loops

- When the loop ends, we will increment by 10
  - *sum += 2*
  - *i++*

```
# sum = 0, i = 0, limit = 5
addi x10, x0, 0    # sum
addi x8, x0, 0    # i
addi x9, x0, 5    # terminating point
```

```
addi x10, x10, 2 #sum +=2
addi x8, x8, 1 # i++
```

```
addi x10, x10, 10 # sum +=10
```

Loops

- Now, let's write the line that will check the terminating condition
- First, we need to add labels to our code that mark the beginning and end of the loop

```
    # sum = 0, i = 0, limit = 5
    addi x10, x0, 0                     # sum
    addi x8, x0, 0                      # i
    addi x9, x0, 5                      # terminating point
LoopStart:
    [redacted]
    addi x10, x10, 2                    #sum +=2
    addi x8, x8, 1                      # i++
    [redacted]
LoopEnd:
    addi x10, x10, 10                   # sum +=10
```

Loops

- We'll use a new instruction called branch on equal (**beq**) to check the terminating condition. The beq instruction in the program below will compare the value of x8 to the value of x9. If they are equal (i.e. i == 5), the program will terminate the loop by jumping to the label, **LoopEnd**. If they are not equal (i.e. i != 5), the program will continue to the next instruction (**addi x10, x10,x 2**).

```
    # sum = 0, i = 0, limit = 5
    addi x10, x0, 0                    # sum
    addi x8, x0, 0                        # i
    addi x9, x0, 5                        # terminating point
LoopStart:
    beq x8, x9 LoopEnd                 # exit loop when i == 5
    addi x10, x10, 2                      #sum +=2
    addi x8, x8, 1                        # i++

    [redacted]
LoopEnd:
    addi x10, x10, 10                     # sum +=10
```

Loops

- We need to add one more instruction that tells the program to jump back to the beginning of the loop once it finishes executing the loop body. This new instruction is called a jump instruction (**`j`**). The jump instruction below tells the program to jump to the line labeled **`LoopStart`**.

```
    # sum = 0, i = 0, limit = 5
    addi x10, x0, 0                    # sum
    addi x8, x0, 0                        # i
    addi x9, x0, 5                        # terminating point
LoopStart:
    beq x8, x9 LoopEnd              # exit loop when i == 5
    addi x10, x10, 2                     #sum +=2
    addi x8, x8, 1                        # i++
    j LoopStart                     # jump to start of loop

LoopEnd:
    addi x10, x10, 10                     # sum +=10
```

Final Program (Pasteable)

```
main:
    # sum = 0, i = 0, limit = 5
    addi x10, x0, 0          # sum = 0
    addi x8,  x0, 0          # i = 0
    addi x9,  x0, 5          # limit = 5

LoopStart:
    beq  x8,  x9, LoopEnd    # exit loop when i == 5
    addi x10, x10, 2         # sum += 2
    addi x8,  x8,  1         # i++
    j    LoopStart           # jump to start of loop

LoopEnd:
    addi x10, x10, 10        # sum += 10 after loop
    jal  x0, halt            # jump to halt
```

Let’s run it!

- Copy the program into our miniRISC-V simulator.
  - *Copy it below the initialization of our stack pointer*
- The following slides explain how to do this.

The C Runtime Startup Code

```
reset:  lui     sp,0xc0000    # initialize stack pointer
        addi    sp,sp,0xff0
        jal     ra,main       # call main
*halt:  j       halt

#########################################
```

This small section of code initializes the stack pointer to the standard 0xbffffff0 and calls the function "main".

This code is loaded into the .kernel memory section.

Note that the lui/addi involves a constant with bit 11 set.

Let's run it!

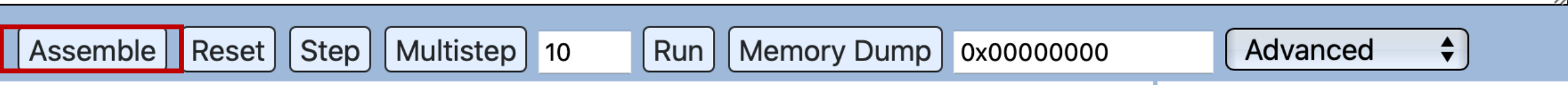
UNC miniRISC-V Architecture Simulator V 1.0

```
reset:  lui     sp,0xc0000   # initialize stack pointer
        addi    sp,sp,0xff0
        jal     ra,main      # call main
*halt:  j       halt

#########################################
main:
    # sum = 0, i = 0, limit = 5
    addi x10, x0, 0          # sum = 0
    addi x8,  x0, 0          # i = 0
    addi x9,  x0, 5          # limit = 5

LoopStart:
    beq  x8,  x9, LoopEnd    # exit loop when i == 5
    addi x10, x10, 2         # sum += 2
    addi x8,  x8,  1         # i++
    j    LoopStart           # jump to start of loop

LoopEnd:
    addi x10, x10, 10        # sum += 10 after loop
    jal  x0, halt            # jump to halt
```

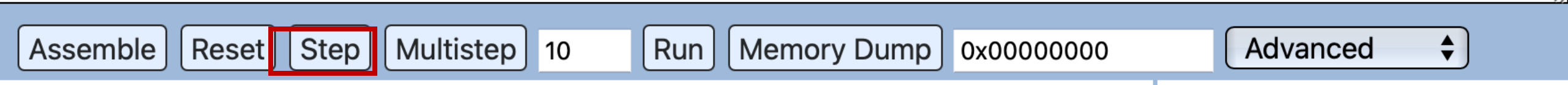
Ways to run your assembly program!

1. Save your work

2. Assemble. Any errors will be displayed.

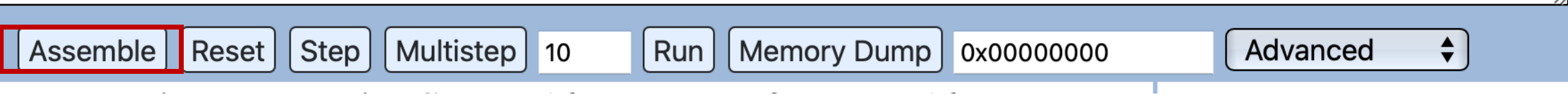


3. Step through a single instruction



Ways to run your assembly program!

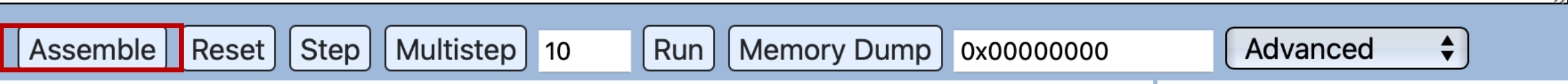
1. Save your work
2. Assemble



3. Step through multiple instructions. You can specify. Default is 10.

Ways to run your assembly program!

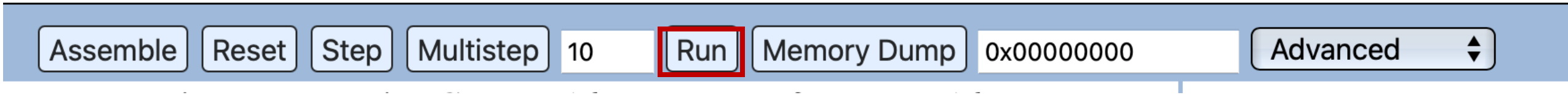
1. Save your work
2. Assemble



Example breakpoint

```
*halt:   j        halt
```

3. Run up until a breakpoint. You can set one by putting a “*” at start of line.



Contents of Registers After Execution

- You can inspect the contents of the register file under the code window
- Each register has a name, according to the ABI

Registers, Instruction Count = 16, Memory References = 16			pc: [0x00000008]
x0/zero: [0x00000000]	x1/ra: [0x0000000C]	x2/sp: [0xBFFFFFF0]	x3/gp: [0x00000000]
x4/tp: [0x00000000]	x5/t0: [0x00000000]	x6/t1: [0x00000000]	x7/t2: [0x00000000]
x8/s0: [0x00000000]	x9/s1: [0x00000000]	x10/a0: [0x00000000]	x11/a1: [0x00000000]
x12/a2: [0x00000000]	x13/a3: [0x00000000]	x14/a4: [0x00000000]	x15/a5: [0x00000000]
x16/a6: [0x00000000]	x17/a7: [0x00000000]	x18/s2: [0x00000000]	x19/s3: [0x00000000]
x20/s4: [0x00000000]	x21/s5: [0x00000000]	x22/s6: [0x00000000]	x23/s7: [0x00000000]
x24/s8: [0x00000000]	x25/s9: [0x00000000]	x26/s10: [0x00000000]	x27/s11: [0x00000000]
x28/t3: [0x00000000]	x29/t4: [0x00000000]	x30/t5: [0x00000000]	x31/t6: [0x00000000]

Contents of Registers After Execution

- You can inspect the contents of the register file in the right panel
- Each register has a name. We'll learn about these names soon.
- We can also see the program counter!

Registers, Instruction Count = 16, Memory References = 16			pc: [0x00000008]
x0/zero: [0x00000000]	x1/ra: [0x0000000C]	x2/sp: [0xBFFFFFF0]	x3/gp: [0x00000000]
x4/tp: [0x00000000]	x5/t0: [0x00000000]	x6/t1: [0x00000000]	x7/t2: [0x00000000]
x8/s0: [0x00000000]	x9/s1: [0x00000000]	x10/a0: [0x00000000]	x11/a1: [0x00000000]
x12/a2: [0x00000000]	x13/a3: [0x00000000]	x14/a4: [0x00000000]	x15/a5: [0x00000000]
x16/a6: [0x00000000]	x17/a7: [0x00000000]	x18/s2: [0x00000000]	x19/s3: [0x00000000]
x20/s4: [0x00000000]	x21/s5: [0x00000000]	x22/s6: [0x00000000]	x23/s7: [0x00000000]
x24/s8: [0x00000000]	x25/s9: [0x00000000]	x26/s10: [0x00000000]	x27/s11: [0x00000000]
x28/t3: [0x00000000]	x29/t4: [0x00000000]	x30/t5: [0x00000000]	x31/t6: [0x00000000]

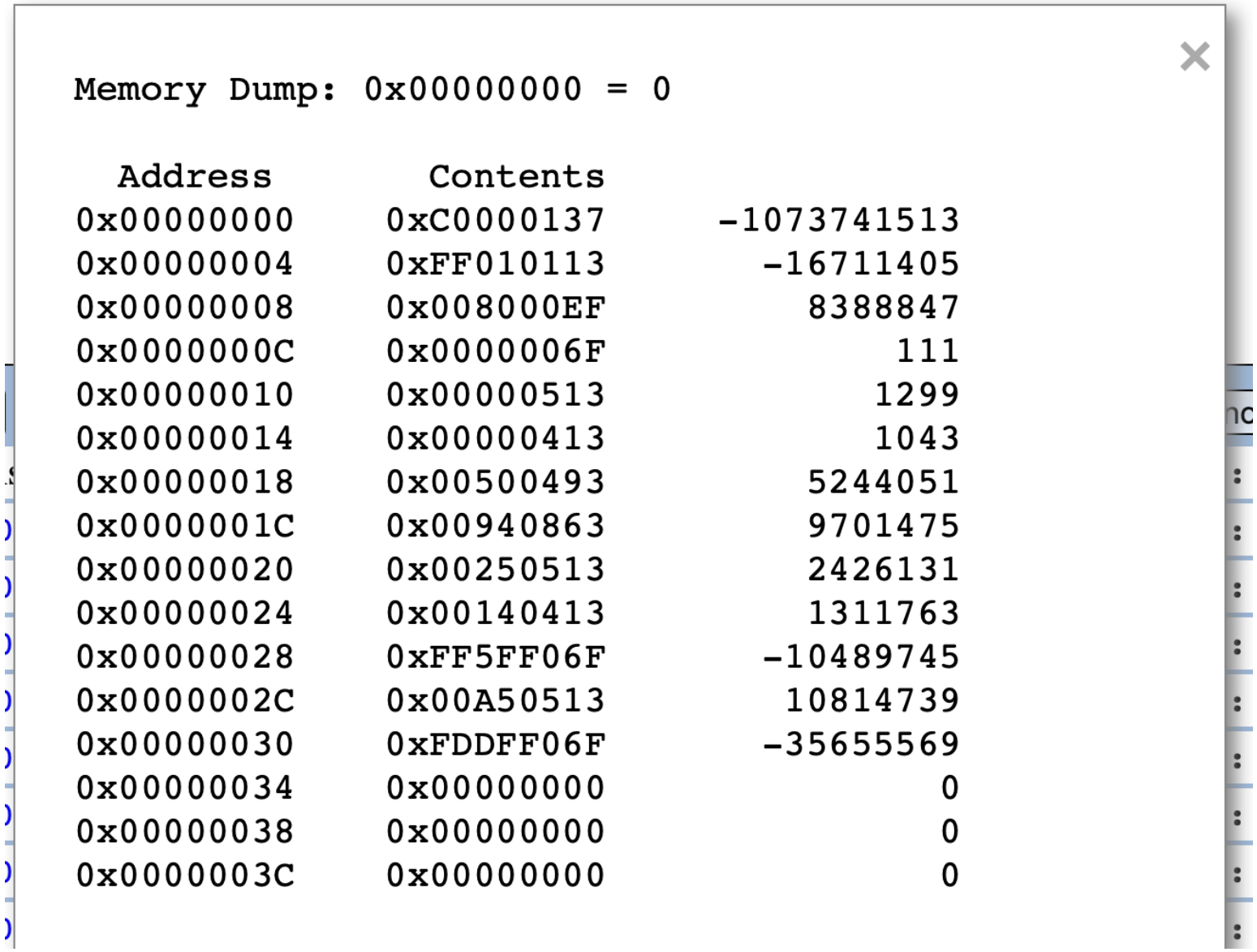
Contents of Registers After Execution

- When the program is finished executing, we can see the final values of the registers. Our program wrote to registers 8, 9, and 10.

Registers, Instruction Count = 29, Memory References = 29			pc: [0x0000000C]
x0/zero: [0x00000000]	x1/ra: [0x0000000C]	x2/sp: [0xBFFFFFF0]	x3/gp: [0x00000000]
x4/tp: [0x00000000]	x5/t0: [0x00000000]	x6/t1: [0x00000000]	x7/t2: [0x00000000]
x8/s0: [0x00000005]	x9/s1: [0x00000005]	x10/a0: [0x00000014]	x11/a1: [0x00000000]
x12/a2: [0x00000000]	x13/a3: [0x00000000]	x14/a4: [0x00000000]	x15/a5: [0x00000000]
x16/a6: [0x00000000]	x17/a7: [0x00000000]	x18/s2: [0x00000000]	x19/s3: [0x00000000]
x20/s4: [0x00000000]	x21/s5: [0x00000000]	x22/s6: [0x00000000]	x23/s7: [0x00000000]
x24/s8: [0x00000000]	x25/s9: [0x00000000]	x26/s10: [0x00000000]	x27/s11: [0x00000000]
x28/t3: [0x00000000]	x29/t4: [0x00000000]	x30/t5: [0x00000000]	x31/t6: [0x00000000]

Memory dump

- You can also view the contents of memory by clicking the memory dump button!



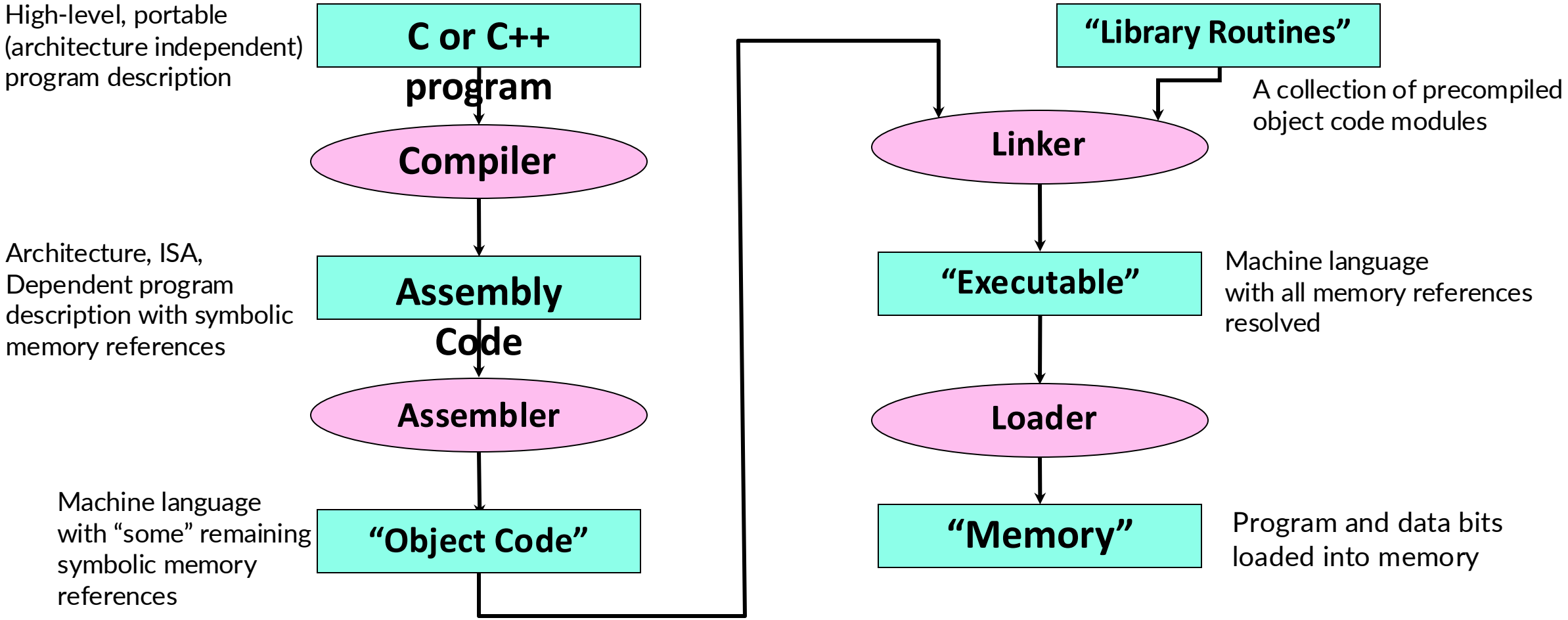
Memory Dump: 0x00000000 = 0

Address	Contents	
0x00000000	0xC0000137	-1073741513
0x00000004	0xFF010113	-16711405
0x00000008	0x008000EF	8388847
0x0000000C	0x0000006F	111
0x00000010	0x00000513	1299
0x00000014	0x00000413	1043
0x00000018	0x00500493	5244051
0x0000001C	0x00940863	9701475
0x00000020	0x00250513	2426131
0x00000024	0x00140413	1311763
0x00000028	0xFF5FF06F	-10489745
0x0000002C	0x00A50513	10814739
0x00000030	0xFDDFF06F	-35655569
0x00000034	0x00000000	0
0x00000038	0x00000000	0
0x0000003C	0x00000000	0

SOME BACKGROUND ON ASSEMBLERS!

The Route from Program to Bits

Traditional Compilation



What an Assembler does

Assembly is just a recipe for sequentially filling memory locations.

```
.word   0x01000293, 0xFFC28293
.word   0x0052AA23, 0xFE029CE3
.word   0x0000006F
.space  4
```

Address	Contents	
0x00000000	0x01000293	16777875
0x00000004	0xFFC28293	-4029805
0x00000008	0x0052AA23	5417507
0x0000000C	0xFE029CE3	-33383197
0x00000010	0x0000006F	111
0x00000014	0x00000000	0
0x00000018	0x00000000	0
0x0000001C	0x00000000	0
0x00000020	0x00000000	0

You can even assemble and run this program

Address	Contents	Instruction
0x00000000	0x01000293	.word 0x01000293, 0xFFC28293 # [addi x5,x0,16]
0x00000004	0xFFC28293	.word 0x01000293, 0xFFC28293 # [addi x5,x5,-4]
0x00000008	0x0052AA23	.word 0x0052AA23, 0xFE029CE3 # [sw x5,20(x5)]
0x0000000C	0xFE029CE3	.word 0x0052AA23, 0xFE029CE3 # [bne x5,x0,.-8]

What an Assembler does

Assembly is just a recipe for sequentially filling memory locations.

```
main:   li      t0,16
loop:   addi    t0,t0,-4
        sw      t0,a(t0)
        bne     t0,x0,loop
halt:   j       halt
a:      .space  4
```

Address	Contents	
0x00000000	0x01000293	16777875
0x00000004	0xFFC28293	-4029805
0x00000008	0x0052AA23	5417507
0x0000000C	0xFE029CE3	-33383197
0x00000010	0x0000006F	111
0x00000014	0x00000000	0
0x00000018	0x00000000	0
0x0000001C	0x00000000	0
0x00000020	0x00000000	0

And this recipe is equivalent to the first

Address	Contents	Instruction
0x00000000	0x01000293	main: li t0,16
0x00000004	0xFFC28293	loop: addi t0,t0,-4
0x00000008	0x0052AA23	sw t0,a(t0)
0x0000000C	0xFE029CE3	bne t0,x0,loop

How an Assembler Works

```
reset:  j       main
array:  .space  11
total:  .word   0

main:   li      t2,0
        li      t3,1
        li      t4,11
        lw      t0,total
        j       test
loop:   add     t0,t0,t3
        slli    t5,t2,2
        sw      t3,array(t5)
        add     t3,t3,t3
        addi    t2,t2,1
test:   blt     t2,t4,loop
        sw      t0,total
*halt:  j       halt
```

So is this

This is an "absolute" address

This is a forward reference

Need to figure out this immediate value

This offset is completely different than the one a few instructions ago

Three components of assembly

1) Allocating and initializing data storage
2) Conversion of mnemonics to binary instructions
3) Resolving addresses

Resolving Addresses- 1st Pass

“Classic” 2-pass assembler approach

Address	Machine Code	Assembly Code
0	0x0000006f	reset: j main
4	0x00000000	array: .space 11
48	0x00000000	total: .word 0
52	0x00000393	main: li t2,0
56	0x00100E13	li t3,1
60	0x00B00E93	li t4,11
64	0x00002283	lw t0,total
68	0x0000006f	j test
72	0x01C282B3	loop: add t0,t0,t3
76	0x00239F13	slli t5,t2,2
80	0x01CF2023	sw t3,array(t5)
84	0x01CE0E33	add t3,t3,t3
88	0x00138393	addi t2,t2,1
92	0x01D3C063	test: blt t2,t4,loop
96	0x00502023	sw t0,total
100	0x0000006F	*halt: j halt

- In the first pass, data and instructions are encoded and assigned offsets, while a symbol table is constructed.
- Unresolved address references are set to 0

Symbol	Location
reset	0
array	4
total	48
main	52
loop	72
test	92
halt	100

Resolving Addresses- 2nd Pass

“Classic” 2-pass assembler approach

Address	Machine Code	Assembly Code
0	0x3400006f	reset: j main
4	0x00000000	array: .space 11
48	0x00000000	total: .word 0
52	0x00000393	main: li t2,0
56	0x00100E13	li t3,1
60	0x00B00E93	li t4,11
64	0x30002283	lw t0,total
68	0x1800006f	j test
72	0x01C282B3	loop: add t0,t0,t3
76	0x00239F13	slli t5,t2,2
80	0x01CF2223	sw t3,array(t5)
84	0x01CE0E33	add t3,t3,t3
88	0x00138393	addi t2,t2,1
92	0xFFD3C6E3	test: blt t2,t4,loop
96	0x02502823	sw t0,total
100	0x0000006F	*halt: j halt

- In the first pass, data and instructions are encoded and assigned offsets, while a symbol table is constructed.
- Unresolved address references are set to 0

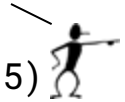
Symbol	Location
reset	0
array	4
total	48
main	52
loop	72
test	92
halt	100

Modern 1-Pass Assembler

Modern assemblers keep more information in their symbol table which allows them to resolve addresses in a single pass.

- Known addresses (backward references) are immediately resolved.
- Unknown or unresolved addresses (forward references) are “back-filled” once they are resolved.

State of the symbol table after the instruction `sw t3, array(t5)` is assembled



Symbol	Address	Resolved?	Reference List
reset	0x00000000 (0)	Y	0
array	0x00000004 (4)	Y	80
total	0x00000030 (48)	Y	64,?
main	0x00000034 (52)	Y	0
loop	0x00000048 (72)	Y	?
test	?	N	68
halt	?	N	?

Role of a Linker

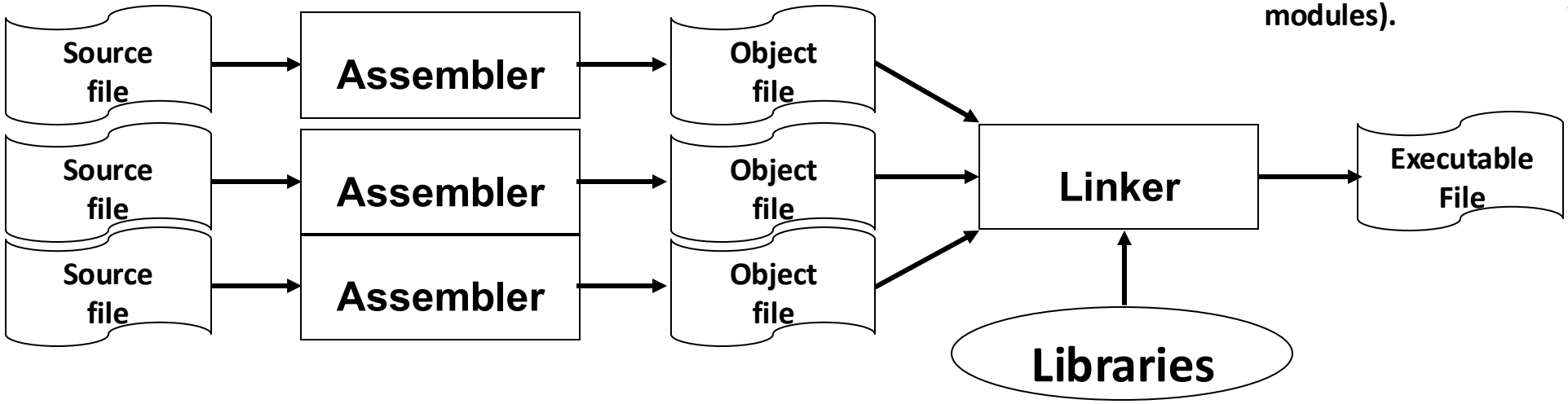
Some aspects of address resolution cannot be handled by the assembler alone.

1. References to data or routines in other object modules
2. The layout of all segments in memory
3. Support for ***REUSABLE*** code modules
4. Support for ***RELOCATABLE*** code modules

This final step of resolution is the job of a **LINKER**

To handle this an object file includes a symbol table with:

1) **Unresolved references**
2) **Addresses of labels declared to be "global" (i.e. accessible to other object modules).**



Static and Dynamic Libraries

- **LIBRARIES** are commonly used routines stored as a concatenation of “Object files”. A global symbol table is maintained for the entire library with **entry points** for each routine.
- When a routine in a LIBRARY is referenced by an assembly module, the routine’s address is resolved by the **LINKER**, and the appropriate code is added to the executable. This sort of linking is called **STATIC** linking.
- Many programs use common libraries. It is wasteful of both memory and disk space to include the same code in multiple executables. The modern alternative to STATIC linking is to allow the **LOADER** and **THE PROGRAM ITSELF** to resolve the addresses of libraries routines. This form of lining is called **DYNAMIC** linking (e.x. .dll).

Dynamically Linked Libraries

- C call to library function:

```
printf("sqr[%d] = %d\n", x, y);
```

- Assembly code

```
        li  a0,#1           # device num of stdio
        li  a1,ctrlstring   # address of "sqr[..."
        lw  a2,x
        lw  a3,y
        auipc  r31,__stdio__
      addi r31,r31,fprintf
        jalr   ra,16(r31)
```

Two things:

1) Calling a function using a pointer
2) There is a table of library entry points located at known fixed offsets from the library's index

How does dynamic linking work?

Dynamically Linked Libraries

"Lazy" address resolution:

```
sysload: addi  sp,sp,-4
         sw        ra,(sp)
         .
         .
         # load stdio library
         .
         .
         # backpatch jump table
         la   t1,__stdio__
         la   t0,dfopen
         sw   t0,(t1)
         la   t0,dfclose
         sw   t0,4(t1)
         la   t0,dfputc
         sw   t0,8(t1)
         la   t0,dfgetc
         sw   t0,12(t1)
         la   t0,dfprintf
         sw   t0,16(t1)
            jalr  ra,(r31)        # we didn't touch it!
            lw    ra,(sp)
            addi   sp,sp,4
            ret
```

Because, the entry points to dynamic library routines are stored in a TABLE. And the contents of this table are loaded on an "as needed" basis!

Before any call is made to a procedure in "stdio.dll"

```
.globl __stdio__:
__stdio__:
fopen:   .word sysload
fclose:  .word sysload
fgetc:   .word sysload
fputc:   .word sysload
fprintf: .word sysload
```

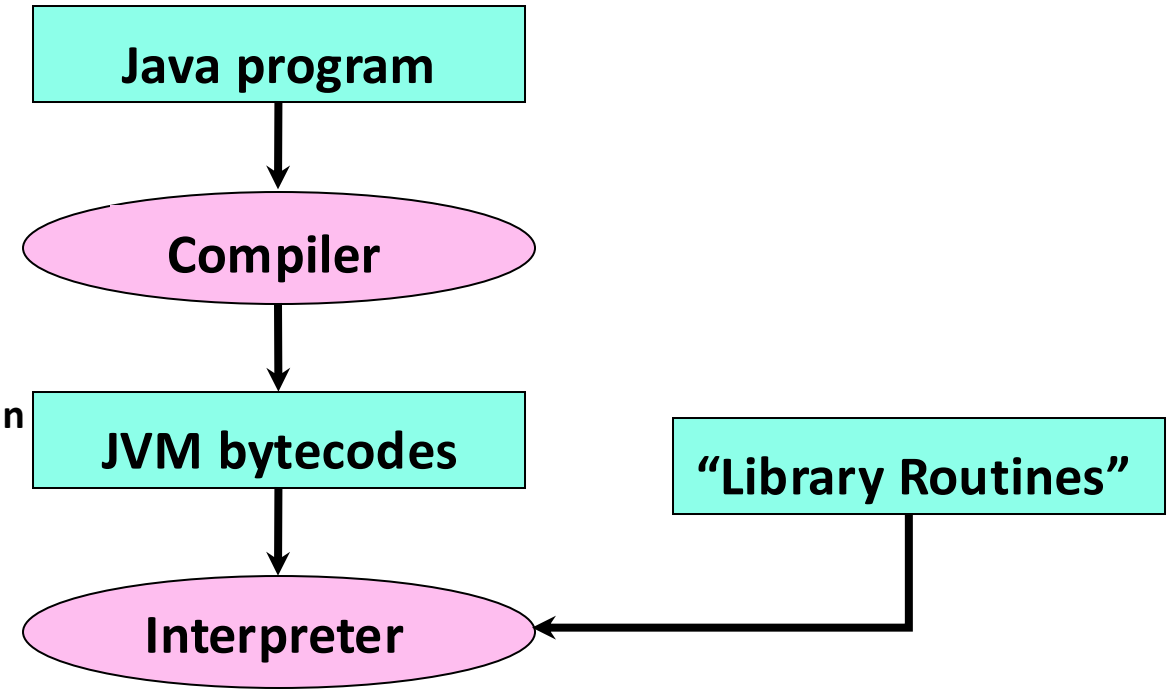
After the first call is made to any procedure in "stdio.dll"

```
.globl __stdio__:
__stdio__:
fopen:   dfopen
fclose:  dclose
fgetc:   dfgetc
fputc:   dfputc
fprintf: dprintf
```

Modern Languages

Intermediate “object code language”

High-level, portable (architecture independent) program description



PORTABLE mnemonic program description with symbolic memory references

An application that EMULATES a virtual machine. Can be written for any Instruction Set Architecture. In the end, machine language instructions must be executed for each JVM bytecode

Modern Languages

Intermediate “object code language”

High-level, portable (architecture independent) program description

Python program

Compiler

PORTABLE abstract machine description with symbolic memory references

Bytecodes (.pyc)

While interpreting on the first pass the JIT keeps a copy of the machine language instructions used.

JIT Complier

Future references access machine language code, avoiding further interpretation

Machine code

Compiled “Library Routines”

This is also how Java works too!

Today’s JITs are nearly as fast as a native compiled code.

Assembly? Really?

- In the early days compilers were dumb
  - *literal line-by-line generation of assembly code of "C" source*
  - *This was efficient in terms of S/W development time*
    - C is portable, ISA independent, write once– run anywhere
    - C is easier to read and understand
    - Details of stack allocation and memory management are hidden
  - *However, a savvy programmer could nearly always generate code that would execute faster*
- Enter the modern era of Compilers
  - *Focused on optimized code-generation*
  - *Captured the common tricks that low-level programmers used*
  - *Meticulous bookkeeping (i.e. will I ever use this variable again?)*
  - *It is hard for even the best hacker to improve on code generated by good optimizing compilers*

Why Assemble when we can Compile?

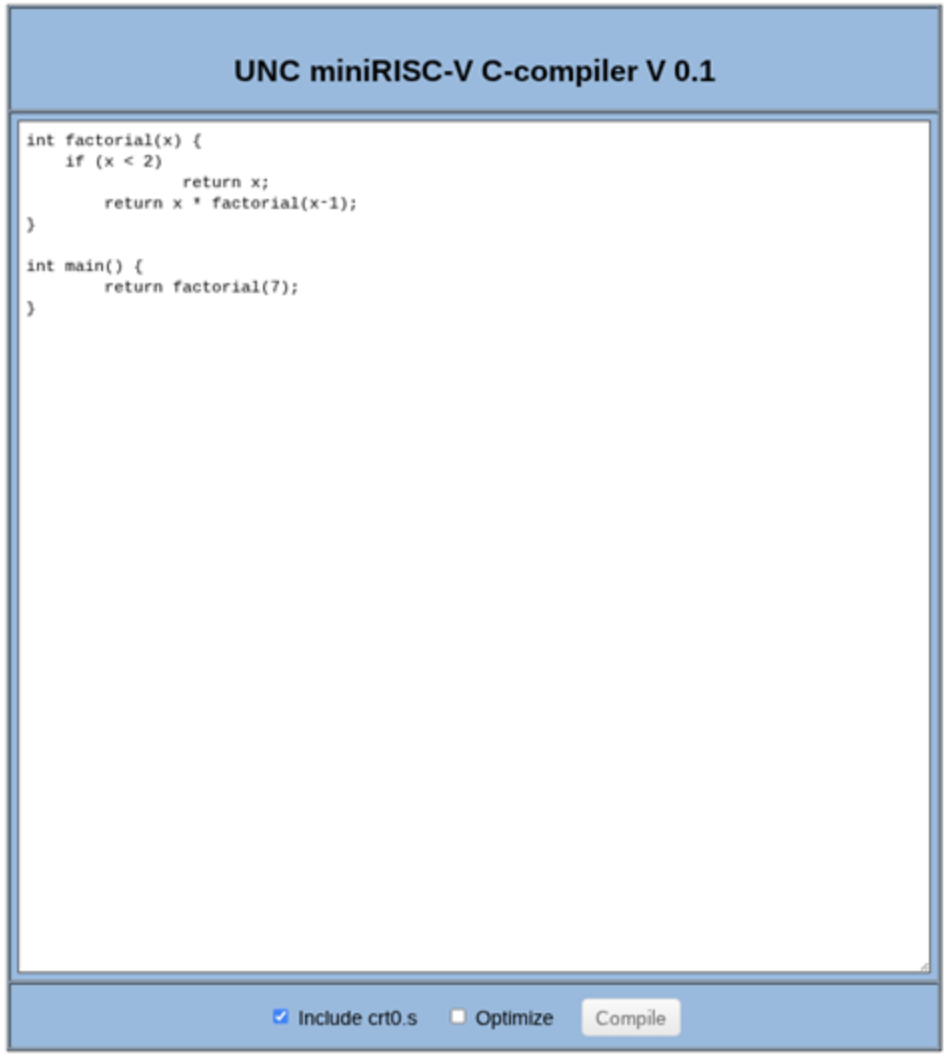
There are a lot of details involved when mapping an assembly language program to bits in memory. But, this mapping can be automated. Compilers provide a means for hiding the details of assembly language.

Lets, revisit our classic recursive program:

```
int factorial(x) {
    if (x < 2)
     return x;
   return x * factorial(x-1);
}

int main() {
   return factorial(7);
}
```

COMPILING TO RISC-V!

We also have a miniRISC-V C-Compiler for you to try!



Goto
https://csbio.unc.edu/mcmillan/index.py?run=rv

This compiler does only code generation. It bypasses the linking stage.

The assembly code that it generates (the "-S" command-line option) can be pasted directly into the simulator

Cut and paste your C-code and press [Compile]

C-Program to Compile

```
int factorial(int x) {
    if (x < 2)
        return x;
    return x * factorial(x - 1);
}

int main() {
    return factorial(7);
}
```

The miniRISC-V C-Compiler

```
        .align 4
        .text
        .globl factorial
        .align  4
factorial:
        addi x2,x2,-64
        sw x8,60(x2)
        addi x8,x2,48
        sw x1,24(x2)
        sw x27,28(x2)
        mv x27,x12
        li x30,2
        bge x27,x30,L.2
        addi x10,x27,0
        jal x0,L.1
L.2:
        addi x12,x27,-1
        jal x1,factorial
        addi x30,x10,0
        mul x10,x27,x30
L.1:
        lw x1,24(x2)
        lw x27,28(x2)
        lw x8,60(x2)
        addi x2,x2,64
        jalr x0,x1,0
```

A larger than necessary stack frame with a "frame pointer" (x8) that is never used.

```
        .globl main
        .align  4
main:
        addi x2,x2,-48
        sw x8,44(x2)
        addi x8,x2,32
        sw x1,24(x2)
        li x12,7
        jal x1,factorial
        addi x30,x10,0
L.4:
        lw x1,24(x2)
        lw x8,44(x2)
        addi x2,x2,48
        jalr x0,x1,0

        .align 4
```

While all the assembly code generated is valid. It does not provide any support for initializing before startup. Rerun the compiler with "include crt0.s" selected.

A "Frame-Pointer"

While the stack pointer (sp) points to the very top of the stack (the most recently allocated item), the frame pointer (fp) provides a stable reference point at the base of the current stack frame.

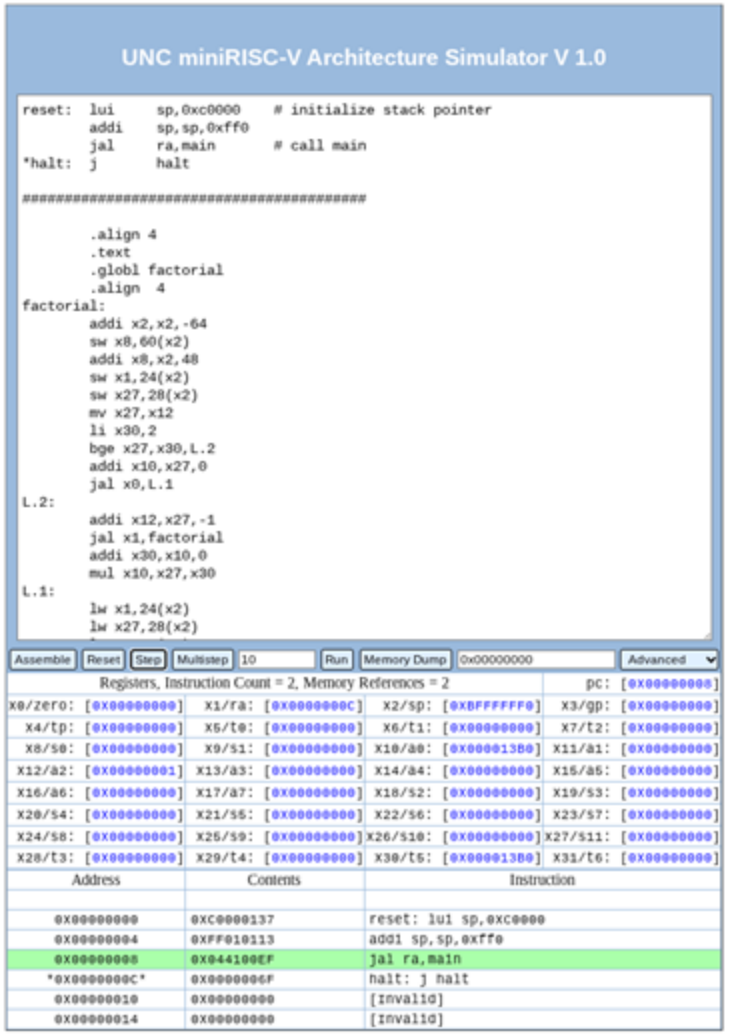
This allows for new local variables to be created with the function without having to adjust the offsets to local variables and other stack values.

```
int amax = 0;
for (int i = 0; i < 10; i++)
    if (a[i] > amax)
        amax = a[i];
```

Stack pointer offsets can vary if new local variables are created within the function, but they are fixed relative to the frame pointer.

Caller's	sp -->	sp+64	
		sp+60	s0/Caller's fp
		sp+56	
		sp+52	
	fp -->	sp+48	
		sp+44	
		sp+40	
		sp+36	
		sp+32	
		sp+28	s11=x
		sp+24	ra
		sp+20	
		sp+16	
		sp+12	
		sp+8	
		sp+4	
	sp -->	sp+0	
			amax
	sp -->		i

Run it!



The code should assemble and run.

The simulator keeps track of the number of instructions executed. (131)

How could we count the number of calls to factorial?

Another Example

Counts Ones:

```
int countOnes(unsigned int x) {
    if (x == 0)
      return x;
    return (x & 1) + countOnes(x>>1);
}

int main() {
   return countOnes(1023);
}
```

What does this fn do?
Compile and test it.

How could you improve it?
Size? Speed?

Without Optimizations

Compiled Assembly Code (Cut and Paste into simulator):

```
reset:  lui     sp,0xc0000      # initialize stack pointer
        addi    sp,sp,0xff0
        jal     ra,main         # call main
*halt:  j       halt

#########################################

        .align 4
        .text
        .globl countOnes
        .align  4
countOnes:
        addi x2,x2,-64
        sw x8,60(x2)
        addi x8,x2,48
        sw x1,24(x2)
        sw x27,28(x2)
        mv x27,x12
        bne x27,x0,L.2
        addi x10,x27,0
        jal x0,L.1
L.2:
        li x30,1
        srl x12,x27,x30
        jal x1,countOnes
        li x29,1
        and x29,x27,x29
        addi x30,x10,0
        add x30,x29,x30
        addi x10,x30,0
L.1:
        lw x1,24(x2)
        lw x27,28(x2)
        lw x8,60(x2)
        addi x2,x2,64
        jalr x0,x1,0

        .globl main
        .align  4
main:
        addi x2,x2,-48
        sw x8,44(x2)
        addi x8,x2,32
        sw x1,24(x2)
        li x12,1023
        jal x1,countOnes
        addi x30,x10,0
L.4:
        lw x1,24(x2)
        lw x8,44(x2)
        addi x2,x2,48
        jalr x0,x1,0

        .align 4
```

Registers, Instruction Count = 228, Memory References = 298

x0/zero: [0x00000000]	x1/ra: [0x0000000C]	x2/sp: [0xBFFFFFF0]	x3/gp: [0x00000000]
x4/tp: [0x00000000]	x5/t0: [0x00000000]	x6/t1: [0x00000000]	x7/t2: [0x00000000]
x8/s0: [0x00000000]	x9/s1: [0x00000000]	x10/a0: [0x0000000A]	x11/a1: [0x00000000]
x12/a2: [0x00000000]	x13/a3: [0x00000000]	x14/a4: [0x00000000]	x15/a5: [0x00000000]
x16/a6: [0x00000000]	x17/a7: [0x00000000]	x18/s2: [0x00000000]	x19/s3: [0x00000000]
x20/s4: [0x00000000]	x21/s5: [0x00000000]	x22/s6: [0x00000000]	x23/s7: [0x00000000]
x24/s8: [0x00000000]	x25/s9: [0x00000000]	x26/s10: [0x00000000]	x27/s11: [0x00000000]
x28/t3: [0x00000000]	x29/t4: [0x00000001]	x30/t5: [0x0000000A]	x31/t6: [0x00000000]

pc: [0x0000000C]

Address	Contents	Instruction
0x00010078	0x02C12403	lw x8,44(x2)
0x0001007C	0x03010113	addi x2,x2,48
0x00010080	0x00008067	jalr x0,x1,0
0x0000000C	0x0000006F	halt: j halt
0x00000010	0x00000000	[Invalid]
0x00000014	0x00000000	[Invalid]
0x00000018	0x00000000	[Invalid]

Does this code adhere to the ABI?