VLSI System Design
(Graduate Level)
Fall 2021

RISC-V Tools

Instructor: Lih-Yih Chiou
Speaker: Willie
Date: 2022.09.07

Outline

Introduction

GNU

RISC-V Tools

Introduction

- The GNU Project
  - Free operating system and software collection
- GNU/Linux
  - GNU OS using Linux kernel
- GNU Toolchain
  - A broad collection of programming tools
- GCC (GNU Compiler Collection)
  - A compiler system supporting various programming languages
- Cross Compiler
  - A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running
  - ARM executable generated from Windows PC
  - RISC-V executable generated from Linux

Introduction

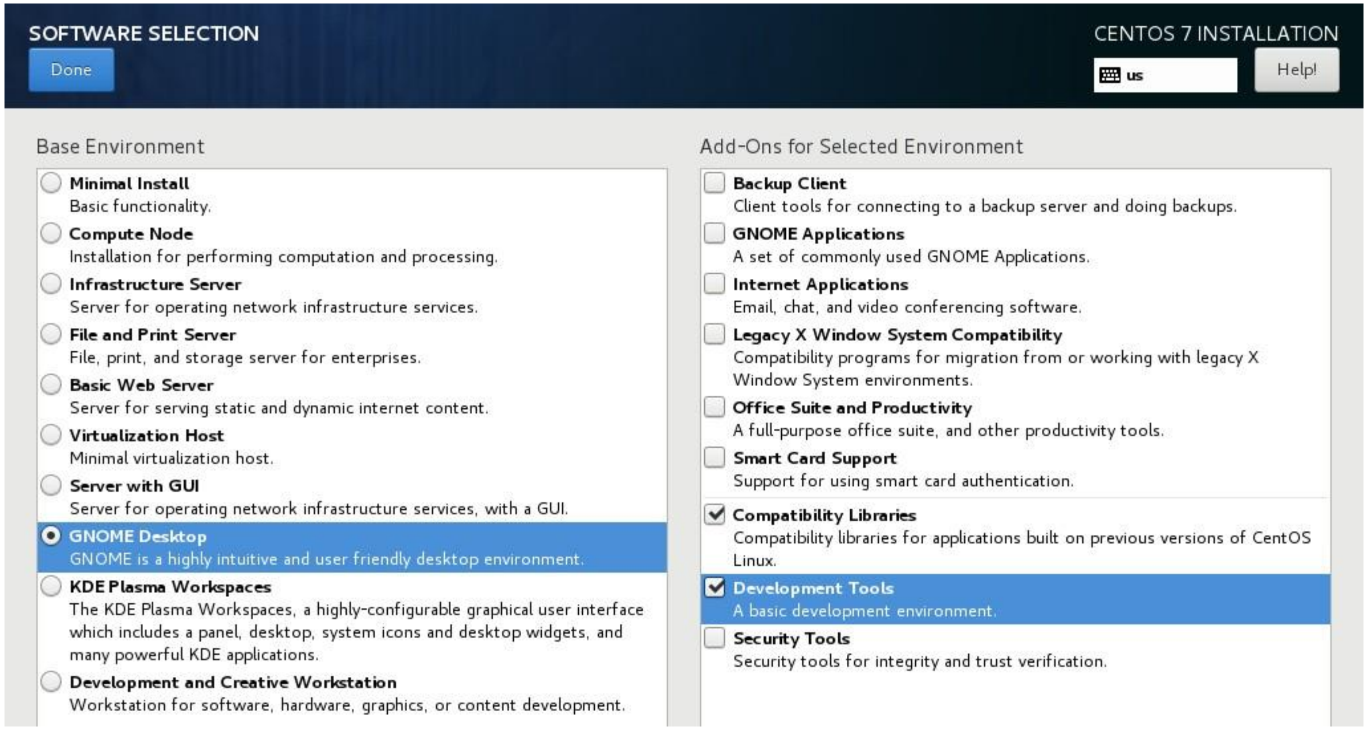
- riscv-gnu-toolchain
  - A RISC-V cross compiler
- riscv-fesvr
  - A “front-end” server that services calls between the host and target processors on the Host-Target InterFace (HTIF)
- riscv-isa-sim
  - The ISA simulator and “golden standard” of execution
- riscv-opcodes
  - The enumeration of all RISC-V opcodes executable by the simulator
- riscv-pk
  - A proxy kernel that services system calls generated by code built and linked with the RISC-V Newlib port
- riscv-tests
  - a set of assembly tests and benchmarks

Installation

CentOS

Installation

- Preparation:
  - CentOS 7 or other Linux OS
  - Development Tools (2 methods)
    - When you install CentOS 7



SOFTWARE SELECTION

Base Environment

- ○ **Minimal Install** Basic functionality.
- ○ **Compute Node** Installation for performing computation and processing.
- ○ **Infrastructure Server** Server for operating network infrastructure services.
- ○ **File and Print Server** File, print, and storage server for enterprises.
- ○ **Basic Web Server** Server for serving static and dynamic internet content.
- ○ **Virtualization Host** Minimal virtualization host.
- ○ **Server with GUI** Server for operating network infrastructure services, with a GUI.
- ● **GNOME Desktop** GNOME is a highly intuitive and user friendly desktop environment.
- ○ **KDE Plasma Workspaces** The KDE Plasma Workspaces, a highly-configurable graphical user interface which includes a panel, desktop, system icons and desktop widgets, and many powerful KDE applications.
- ○ **Development and Creative Workstation** Workstation for software, hardware, graphics, or content development.

Add-Ons for Selected Environment

- ☐ **Backup Client** Client tools for connecting to a backup server and doing backups.
- ☐ **GNOME Applications** A set of commonly used GNOME Applications.
- ☐ **Internet Applications** Email, chat, and video conferencing software.
- ☐ **Legacy X Window System Compatibility** Compatibility programs for migration from or working with legacy X Window System environments.
- ☐ **Office Suite and Productivity** A full-purpose office suite, and other productivity tools.
- ☐ **Smart Card Support** Support for using smart card authentication.
- ✓ **Compatibility Libraries** Compatibility libraries for applications built on previous versions of CentOS Linux.
- ✓ **Development Tools** A basic development environment.
- ☐ **Security Tools** Security tools for integrity and trust verification.

```
$ yum -y groupinstall “Development Tools”
```

Installation

- Preparation:
  - Required packages

```
$ yum -y upgrade
$ yum -y install epel-release
$ yum -y install dtc libmpc-devel texinfo gperf zlib-devel expat-devel
```

  - RISC-V toolchain

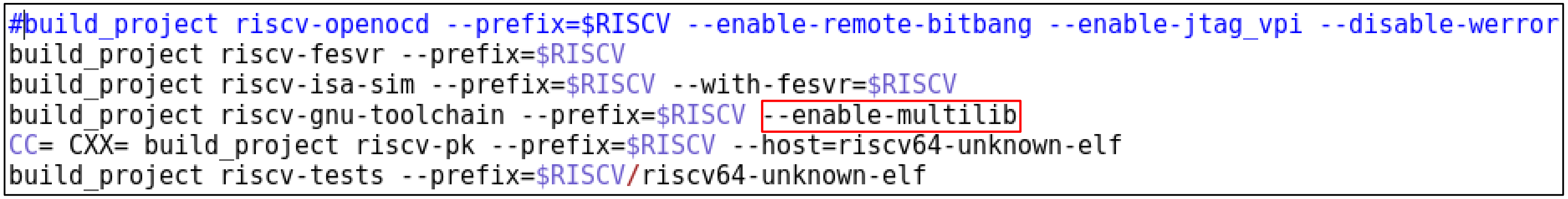
```
$ git clone https://github.com/riscv/riscv-tools.git /opt/RISCV/riscv-tools
$ cd /opt/RISCV/riscv-tools
$ git submodule update --init --recursive
```

Installation

- Setup and install RISC-V tools
  - Edit installation setting

    $ gedit build.sh

    - Add a pound sign (#) before build_project riscv-openocd to build without OpenOCD
    - Add *--enable-multilib* in the end of build_project riscv-gnu-toolchain



```
#build_project riscv-openocd --prefix=$RISCV --enable-remote-bitbang --enable-jtag_vpi --disable-werror
build_project riscv-fesvr --prefix=$RISCV
build_project riscv-isa-sim --prefix=$RISCV --with-fesvr=$RISCV
build_project riscv-gnu-toolchain --prefix=$RISCV --enable-multilib
CC= CXX= build_project riscv-pk --prefix=$RISCV --host=riscv64-unknown-elf
build_project riscv-tests --prefix=$RISCV/riscv64-unknown-elf
```

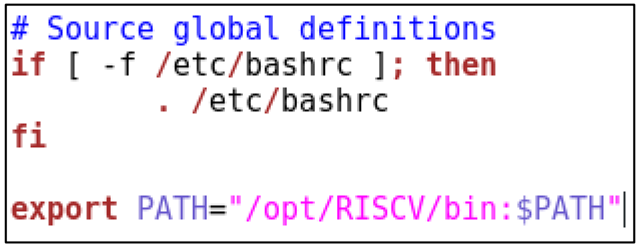
    - Save and exit
  - Install RISC-V tools

    $ RISCV=/opt/RISCV bash build.sh
  - Setup environment variables

    $ gedit ~/.bashrc

    - Add export PATH="/opt/RISCV/bin:$PATH" at EOF
    - Save and exit

    $ source ~/.bashrc



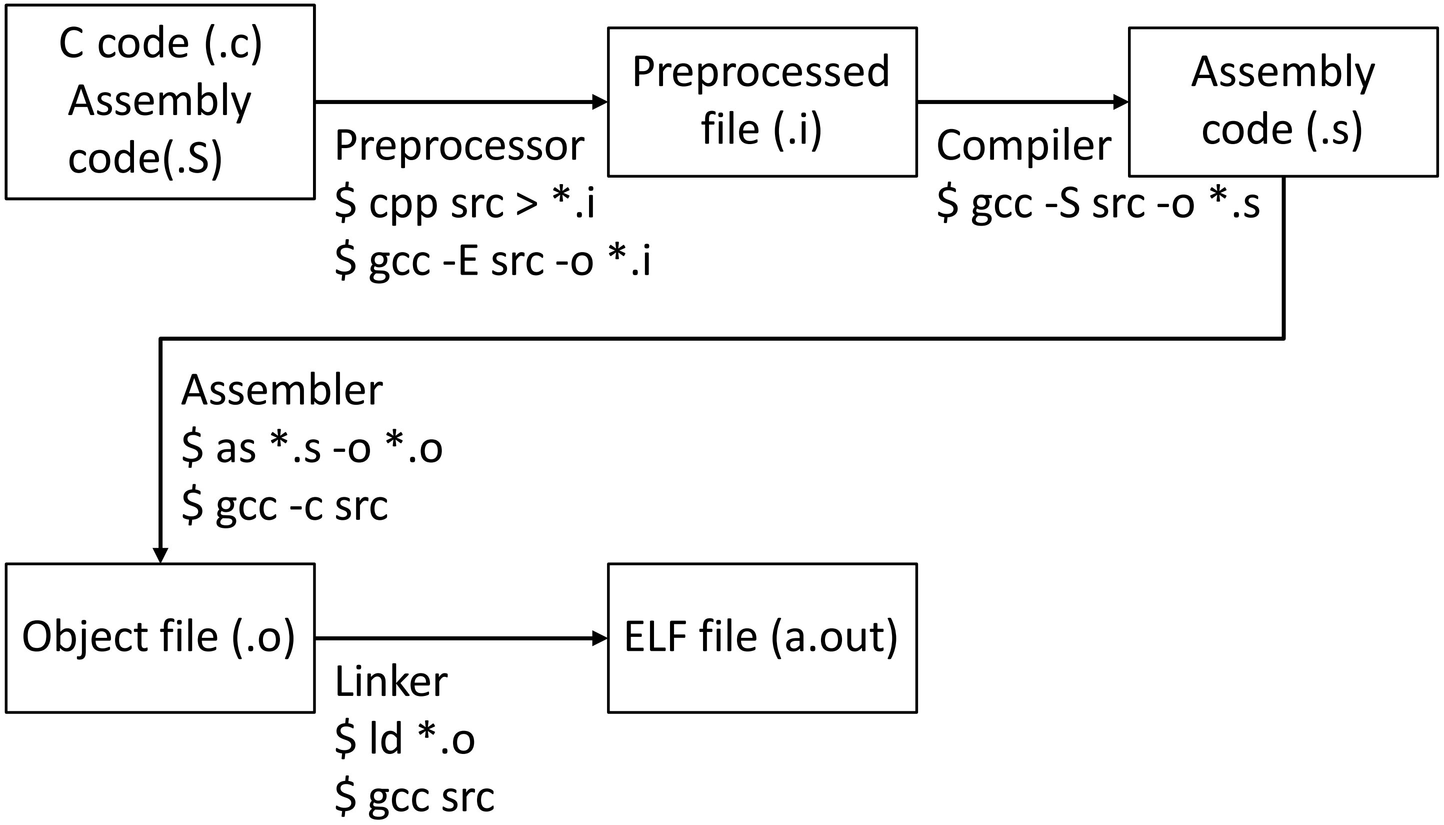
```
# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

export PATH="/opt/RISCV/bin:$PATH"
```

Tools Usage

GCC

Spike (RISC-V ISA Simulator)

GCC Compilation Procedure



Preprocessor

```
$ cpp src > *.i
$ gcc -E src -o *.i
```

Compiler

```
$ gcc -S src -o *.s
```

Assembler

```
$ as *.s -o *.o
$ gcc -c src
```

Linker

```
$ ld *.o
$ gcc src
```

RISC-V GNU Compiler Collection

- Write a “Hello World” program

```
$ mkdir -p ~/hello && cd ~/hello
$ gedit hello.c
```

```c
#include <stdio.h>
int main(void) {
  printf("Hello world!\n");
  return 0;
}
```

- Generate the assembly code

```
$ riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -S hello.c
```

- Generate the object file

```
$ riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -c hello.c # hello.i / hello.s
```

- Generate the ELF file

```
$ riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 hello.c -o hello # hello.i /
  hello.s / hello.o
```

- Generate the log file

```
$ riscv64-unknown-elf-objdump -xsd hello > hello.log
```

- Generate the Verilog hex file

```
$ riscv64-unknown-elf-objcopy hello -O verilog hello.hex
```

Spike

- Run the executable file (ELF) using proxy kernel and boot loader
  - # spike pk hello # Hello world!
- Running debug mode
  - # spike -d pk hello
  - Advance by one instruction
    - Press Enter
  - Continue execution indefinitely
    - : r
  - See the content of an integer register (0 means core 0)
    - : reg 0 t0
  - See the content of a memory location (physical address in hex)
    - : mem 1000
  - See the content of memory with a virtual address (0 for core 0)
    - : mem 0 1000
  - Execute until the condition is reached
    - : until pc 0 1010   (pc of core 0 is equal to 0x1010)
    - : until mem 80011e2c 80003f1c
  - Quit debug mode
    - : q

RISC-V Linker Script

Linker Script

Section

Linker Script

- Assume that IM and DM are using unified memory. Each of their size is 32KB
  - Actually, they aren't using unified memory. This situation will be fixed when you add bus to the system.

```
OUTPUT_ARCH( "riscv" )

_STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE : 0x1000;
_TEST_SIZE = DEFINED(_TEST_SIZE) ? _TEST_SIZE : 0x100;

/*****************************************************
 * Define memory layout
 *****************************************************
MEMORY {
  imem : ORIGIN = 0x00000000, LENGTH = 0x00008000
  dmem : ORIGIN = 0x00008000, LENGTH = 0x00008000
}
```

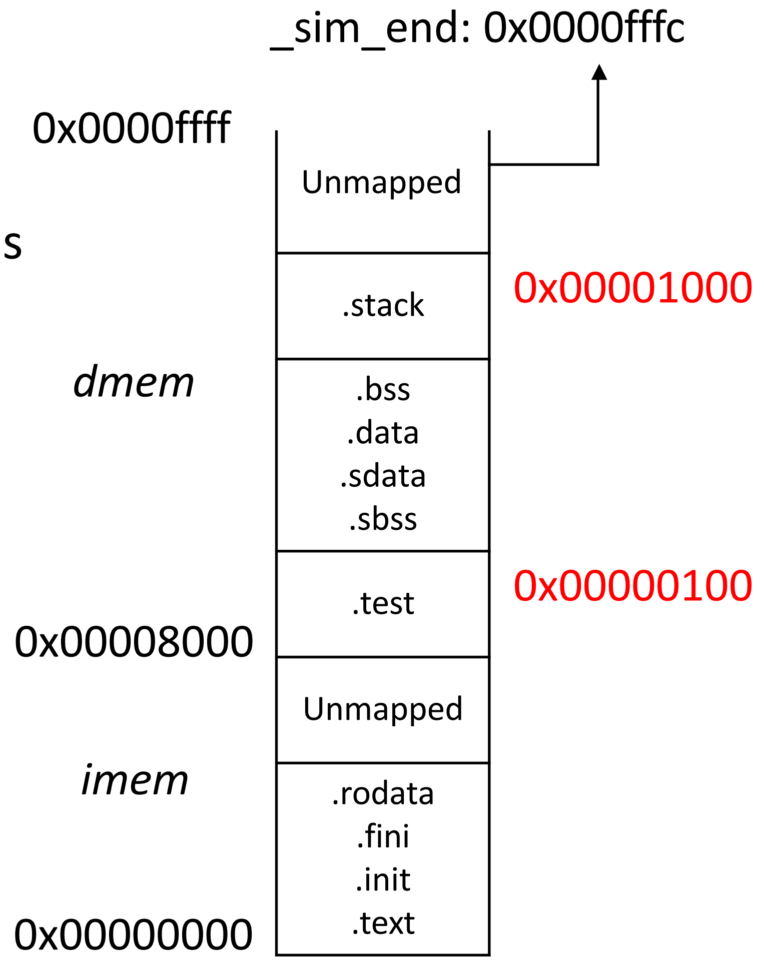
```
SECTIONS {
  .text : {
    ...
  } > imem

  .init : {
    ...
  } > imem

  .fini : {
    ...
  } > imem

  .rodata : {
    ...
  } > imem

  _test : {
    ...
  } > dmem
```

```
  .sbss : {
    ...
  } > dmem

  .sdata : {
    ...
  } > dmem

  .data : {
    ...
  } > dmem

  .bss : {
    ...
  } > dmem

  .stack : {
    ...
  } > dmem

  ...
  _end = .;
}
```

Linker Script

- Assume that IM and DM are using unified memory. Each of their size is 32KB
  - Actually, they aren't using unified memory. This situation will be fixed when you add bus to the system.

```
OUTPUT_ARCH( "riscv" )

_STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE : 0x1000;
_TEST_SIZE = DEFINED(_TEST_SIZE) ? _TEST_SIZE : 0x100;

/**********************************************************
 * Define memory layout
 **********************************************************
MEMORY {
  imem : ORIGIN = 0x00000000, LENGTH = 0x00008000
  dmem : ORIGIN = 0x00008000, LENGTH = 0x00008000
}
```



Sections

- .text
  - Store instruction code
- .init & .fini
  - Store instruction code for entering & leaving the process
- .rodata
  - Store constant global variable
- .bss & .sbss
  - Store uninitiated global variable or global variable initiated as zero
- .data & .sdata
  - Store global variable initiated as non-zero
- .stack
  - Store local variables
- Test by yourself

```
$ cd section && make && riscv64-unknown-elf-objdump -xd main.o
```

RISC-V Assembly Code

Assembly Format

Pseudo-Ops

Labels & Pseudo-Instructions

RISC-V Assembly Code

Available registers

The RISC-V Instruction Set Manual
Volume I: User-Level ISA
Document Version 2.2
Chapter 20
RISC-V Assembly Programmer's Handbook

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Assembly format

Type	Format	Example	Explanation
R-type	inst rd, rs1, rs2	add t0, t1, t2	t0 = t1 + t2
I-type	inst rd, rs1, imm	addi t0, t0, 4	t0 = t0 + 4
Load	inst rd, imm(rs1)	lw t0, 0(sp)	t0 = MEM[sp + 0]
S-type	inst rs2, imm(rs1)	sw t0, 0(sp)	MEM[sp + 0] = t0
B-type	inst rs1, rs2, imm	blt t0, t1, 16	PC = $(t0 < t1)_s$? PC +16: PC + 4
U-type	Inst rd, imm	lui t0, 0x40000	t0 = 0x4000_0000
J-type	inst rd, imm	jal ra, 16	ra = PC + 4; PC = PC + 16

Pseudo-Ops

- .align
  - Align to power of 2 (“.align 2” means next code or data start at MEM[4n])
- .globl
  - Emit symbol_name to symbol table (scope GLOBAL)
  - The symbol can be found by other files
- .section
  - Define following data’s or code’s section (.section [{.text, .data, .rodata, .bss}])
- .equ
  - Constant definition (.equ abc 0x1000)
- Data type
  - .string, .byte (8-bit), .half (16-bit), .word (32-bit), .dword (64-bit)

Labels & Pseudo-Instructions

- Label
  - Text labels are added to the symbol table
  - Numeric labels are used for local references
- Load Immediate
  - li a0, 0x12345678
- Load Address
  - la a1, loop

```
loop:
  j loop

=================

1:
  j 1b
```

```
  j end
  ...
end:
  ...

=================

  j 1f
  ...
1:
  ...
```

The RISC-V Instruction Set Manual
Volume I: User-Level ISA
Document Version 2.2
Chapter 20
RISC-V Assembly Programmer's Handbook

Write Code for Homework

Setup Environment

Assembly Code

C Code

Setup Environment (setup.S)

- Reset Register File (→)
- Initial Pointers & Sections (↓)

```
 /* initialize global pointer */
 la gp, _gp

init_bss:
  /* init bss section */
  la a0, __bss_start
  la a1, __bss_end-4 /* section end is actually the start of the next section */
  li a2, 0x0
  jal fill_block

init_sbss:
  /* init bss section */
  la a0, __sbss_start
  la a1, __sbss_end-4 /* section end is actually the start of the next section */
  li a2, 0x0
  jal fill_block

write_stack_pattern:
  /* init stack section */
  la a0, _stack_end  /* note the stack grows from top to bottom */
  la a1, __stack-4   /* section end is actually the start of the next section */
  li a2, 0x0
  jal fill_block

init_stack:
  /* set stack pointer */
  la sp, _stack

write_test_pattern:
  la a0, _test_start
  la a1, _test_end-4
  li a2, 0x0
  jal fill_block
```

```
/* Fills memory blocks */
fill_block:
  sw a2, 0(a0)
  bgeu a0, a1, fb_end
  addi a0, a0, 4
  j fill_block
fb_end:
  ret
```

```
_start:
 li x1, 0
 li x2, 0
 li x3, 0
 li x4, 0
 li x5, 0
 li x6, 0
 li x7, 0
 li x8, 0
 li x9, 0
 li x10, 0
 li x11, 0
 li x12, 0
 li x13, 0
 li x14, 0
 li x15, 0
 li x16, 0
 li x17, 0
 li x18, 0
 li x19, 0
 li x20, 0
 li x21, 0
 li x22, 0
 li x23, 0
 li x24, 0
 li x25, 0
 li x26, 0
 li x27, 0
 li x28, 0
 li x29, 0
 li x30, 0
 li x31, 0
```

```
.sdata : {
  _gp = . + 0x800;
  *(.srodata.cst16)
  *(.sdata .sdata.*
} > dmem
```

```
.stack : {
  . = ALIGN(4);
  _stack_end = .;
  . += _STACK_SIZE;
  _stack = .;
  __stack = _stack;
} > dmem
```

Global and stack pointer in linker script

Setup Environment (setup.S)

- Jump to main function
- Wait main function return and then store end code to *_sim_end*

```
SystemInit:
  jal main

SystemExit:
  /* End simulation */
  la t0, _sim_end
  li t1, -1
  sw t1, 0(t0)
dead_loop:
  j dead_loop
```

Problem Description

- Write a program to perform addition. The augend is stored at the address named add*1* in ".rodata" section defined in *data.S*. The addend is stored at the address named add*2* in ".rodata" section defined in *data.S*. The augend and the addend are **signed 4-byte integers**. Their sum is **signed 4-byte integers** and should be stored at the address named *_test_start* in "_test" section defined in *link.ld*.

```
# Define constants
.section .rodata
.align 2
.global add1
.global add2
add1:
  .word 0xffffffff
add2:
  .word 0xffffffff
```

```
  _test : {
    . = ALIGN(4);
    _test_start = .;
    . += _TEST_SIZE;
    _test_end = .;
  } > dmem
```

data.S

"_test" section defined in link.ld

Using Assembly Code

- Load address of add1 & add2
- Add them and store to _test_start

$ cd asm && make

```
.section .text
.align 2
.globl main
main:
  lw t1, add1
  lw t2, add2
  add t1, t1, t2
  sw t1, _test_start, t0
  ret
```

```
00000118 <main>:
 118:   00000317                auipc   t1,0x0
 11c:   02032303                lw      t1,32(t1) # 138 <__rodata_start>
 120:   00000397                auipc   t2,0x0
 124:   01c3a383                lw      t2,28(t2) # 13c <add2>
 128:   00730333                add     t1,t1,t2
 12c:   00008297                auipc   t0,0x8
 130:   ec62aa23                sw      t1,-300(t0) # 8000 <_test_start>
 134:   00008067                ret
```

l{b|h|w|d} rd, symbol	auipc rd, symbol[31:12] l{b|h|w|d} rd, symbol[11:0](rd)	Load global
s{b|h|w|d} rd, symbol, rt	auipc rt, symbol[31:12] s{b|h|w|d} rd, symbol[11:0](rt)	Store global
ret	jalr x0, x1, 0	Return from subroutine

Using C Code

- Load address of add1 & add2
- Add them and store to _test_start

$ cd c && make

```c
int main(void) {
  extern int add1;
  extern int add2;
  extern int _test_start;

  *(&_test_start) = add1 + add2;

  return 0;
}
```

```
00000118 <main>:
 118:   ff010113                addi    sp,sp,-16
 11c:   00812623                sw      s0,12(sp)
 120:   01010413                addi    s0,sp,16
 124:   14c02703                lw      a4,332(zero) # 14c <__rodata_start>
 128:   15002783                lw      a5,336(zero) # 150 <add2>
 12c:   00f70733                add     a4,a4,a5
 130:   000087b7                lui     a5,0x8
 134:   00e7a023                sw      a4,0(a5) # 8000 <_test_start>
 138:   00000793                li      a5,0
 13c:   00078513                mv      a0,a5
 140:   00c12403                lw      s0,12(sp)
 144:   01010113                addi    sp,sp,16
 148:   00008067                ret
```

Reference

Reference

Reference

- https://riscv.org/software-tools/
- https://en.wikipedia.org/wiki/GNU_toolchain
- https://en.wikipedia.org/wiki/Cross_compiler
- https://codingfreak.blogspot.com/2008/02/compilation-process-in-gcc.html
- https://resources.infosecinstitute.com/hello-world-c-assembly-object-file-and-executable/
- https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md

Appendix

Install RISC-V OpenOCD

Install RISC-V OpenOCD

How to install riscv-openocd

```
$ yum install -y libusb-devel libusbx-devel centos-release-scl
$ rpm -ivh
  https://www.softwarecollections.org/en/scls/praiskup/autotools/epel-7-
  x86_64/download/praiskup-autotools-epel-7-x86_64.noarch.rpm
$ yum install -y autotools-latest
$ scl enable autotools-latest bash
$ cd riscv-openocd
$ aclocal
$ autoreconf -i -I /usr/share/aclocal/
$ autoconf
$ autoheader
$ automake --add-missing
$ cd ..
$ gedit build.sh # Uncomment riscv-openocd part
$ bash build.sh
```

Install RISC-V OpenOCD (CentOS 6)

- How to install riscv-openocd

```
$ yum install -y libusb-devel libusb1-devel centos-release-scl
$ rpm -ivh
  https://www.softwarecollections.org/en/scls/praiskup/autotools/epel-6-
  x86_64/download/praiskup-autotools-epel-6-x86_64.noarch.rpm
$ yum install -y autotools-latest devtoolset-6-toolchain
$ scl enable autotools-latest bash
$ cd riscv-openocd
$ aclocal
$ ln -s /usr/share/aclocal/pkg.m4
  /opt/rh/autotools-latest/root/usr/share/aclocal/pkg.m4
$ autoreconf -i
$ autoconf
$ autoheader
$ automake --add-missing
$ cd ..
$ gedit build.sh # Uncomment riscv-openocd part
$ echo "RISCV=/opt/RISCV bash build.sh" | scl enable devtoolset-6 -
```