RISC-V

System-on-Chip Design

Harris, Stine, Thompson, & Harris

Chapter 3:

RISC-V Software Tool Flow

Chapter 3 :: Topics

RISC-V Introduction

3.1 Platforms & Tools
3.2 Getting Started
3.3 GCC Assembler
3.4 GCC Compiler
3.5 QEMU Simulation
3.6 Test Suites

Intro to the RISC-V Software Tools

The RISC-V toolchain includes:

- **GCC (GNU Compiler Collection)**
  - Compiler (C to RISC-V assembly)
  - Assembler (assembly to machine code)
  - Linker (group multiple files and libraries into an executable)
  - Disassembler
- **Other features**
  - Toolchain conforms to the application binary interface (ABI)
  - Spike instruction set simulator (ISS)

The ***RISC-V toolchain*** is used to compile, test, and debug programs targeted to RISC-V systems.

Compile can also refer to all three steps of compiling, assembling, and linking.

Additional Tool and Test Suites:

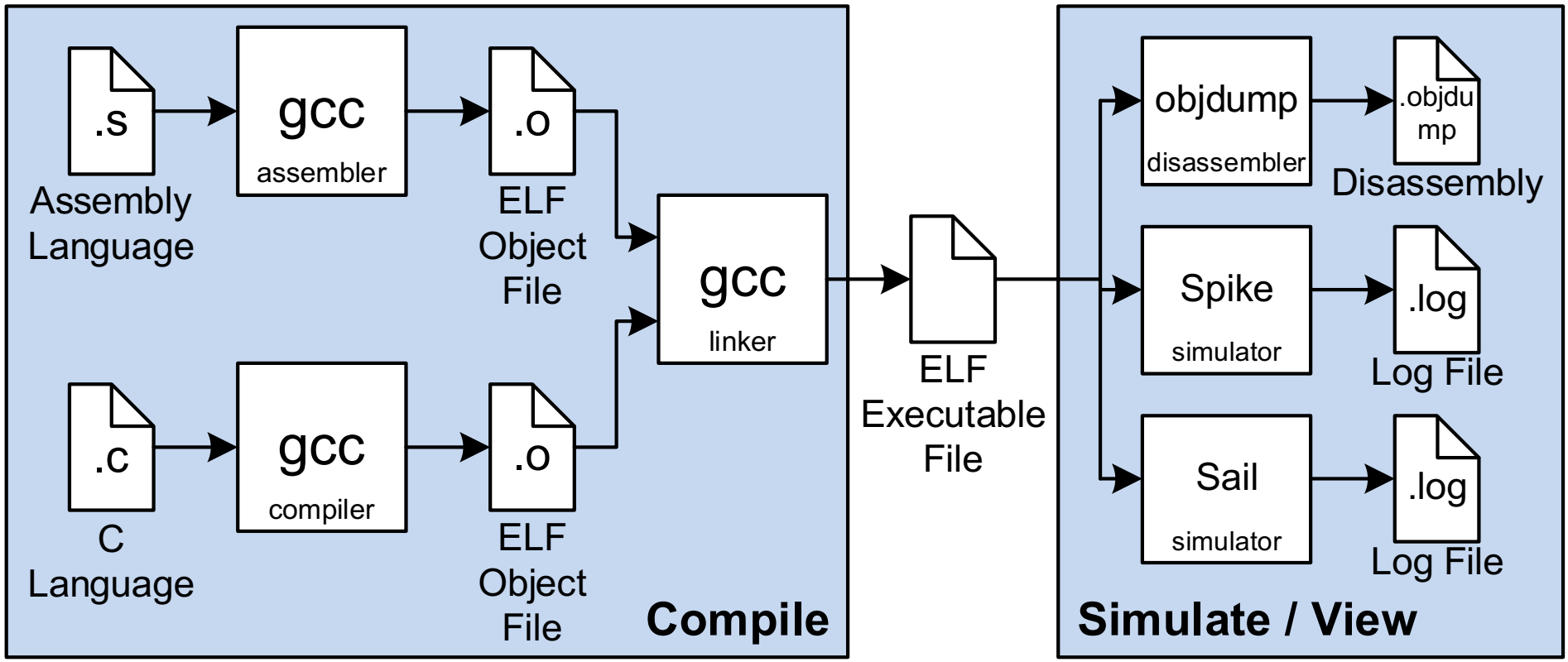
- **QEMU Emulator**
  - Emulates hardware features such as peripherals
- **Test Suites**
  - Group of programs to exercise and verify various features

Tools for using Wally

Chapter	Purpose	Platform / Tool
	Operating System	Linux (Ubuntu Desktop Linux 20.04 LTS, Red Hat Enterprise Linux 8, Rocky Linux 8)
3	**Compiler**	GCC
	Instruction Set Simulator (ISS)	Spike, Sail
	Software Emulator	QEMU
	Test Suites	riscv-arch-tests, Wally-specific
4	**Hardware Simulator**	Questa (Siemens), Verilator (open)
	Hardware Synthesis	Design Compiler (Synopsys)

Questa and Design Compiler require licenses. They are expensive and only available to companies and universities, not individuals. **The other tools are free.**

RISC-V Software Tool Flow



Compile source files into ELF executable

Simulate program running (Spike, Sail) or View compiled program (objdump)

Chapter 3: RISC-V Software Tool Flow

Getting Started

Wally & Tool Installation

- **Installation instructions in README at:**
  https://github.com/openhwgroup/cvw

- **May install tools as:**
  - **Individual user** on a personal Linux computer
    - Must have administrator access
    - User installs tools and clones Wally repository into home directory
  - **On a shared Linux server** (most common case)
    - System administrator installs tools
    - Users clone Wally repository into home directory

Environment Variables

- **When setting up Wally & tools:**
  - **$RISCV:** shared RISC-V tools (typically in /opt/riscv)
  - **$WALLY:** Wally repository in home directory (typically in /home/<username>/cvw)
  - These are set as shown in the cvw README

Chapter 3: RISC-V Software Tool Flow

GCC Assembler

Intro to GCC

- **GCC (GNU Compiler Collection):**
  - High-quality, open-source tool suite (also called *toolchain*)
  - Includes cross-compilers for RISC-V: use one architecture (i.e., x86) to build programs that target another architecture (i.e., RISC-V)
  - Includes:
    - **C compiler** (converts C code to assembly): creates .S file
    - **Assembler** (converts assembly to machine code): creates .o file
    - **Linker** (combines .o and library files to create executable)
    - **Debugger** (gdb)
    - **C library** (newlib or glibc – this setup uses newlib, a lightweight C library)
    - **Binary utilities** for viewing binary files (object files and executables), including:
      - **objdump:** disassembler (converts object file to assembly)

Richard Stallman

- Pioneer of the free software movement
- Founding developer of the GNU operating system and Free Software Foundation (FSF).
- Attended Harvard and became a programmer at the MIT Artificial Intelligence Laboratory.
- Primary developer of GCC, the GDB debugger, make, and the emacs text editor.
- After denouncing proprietary software as vicious and unjust, he defined the four essential freedoms of free software:

  0. The freedom to **run the program as you wish**, for any purpose.
  1. The freedom to **study** how the program works and **change** it so it does your computing as you wish.
  2. The freedom to **redistribute** copies so you can help your neighbor.
  3. The freedom to distribute copies of your **modified versions** to others. By doing this you can give the whole community a chance to benefit from your changes.

  [Stallman, Richard, "GNU's Bulletin, Volume 1 Number 1," gnu.org, Feb. 1986.]

A Quick RISC-V Assembler Tutorial

Steps:

1. **Assemble** .S (assembly) file to create executable file

   ```
   gcc –o <filename> ...
   ```

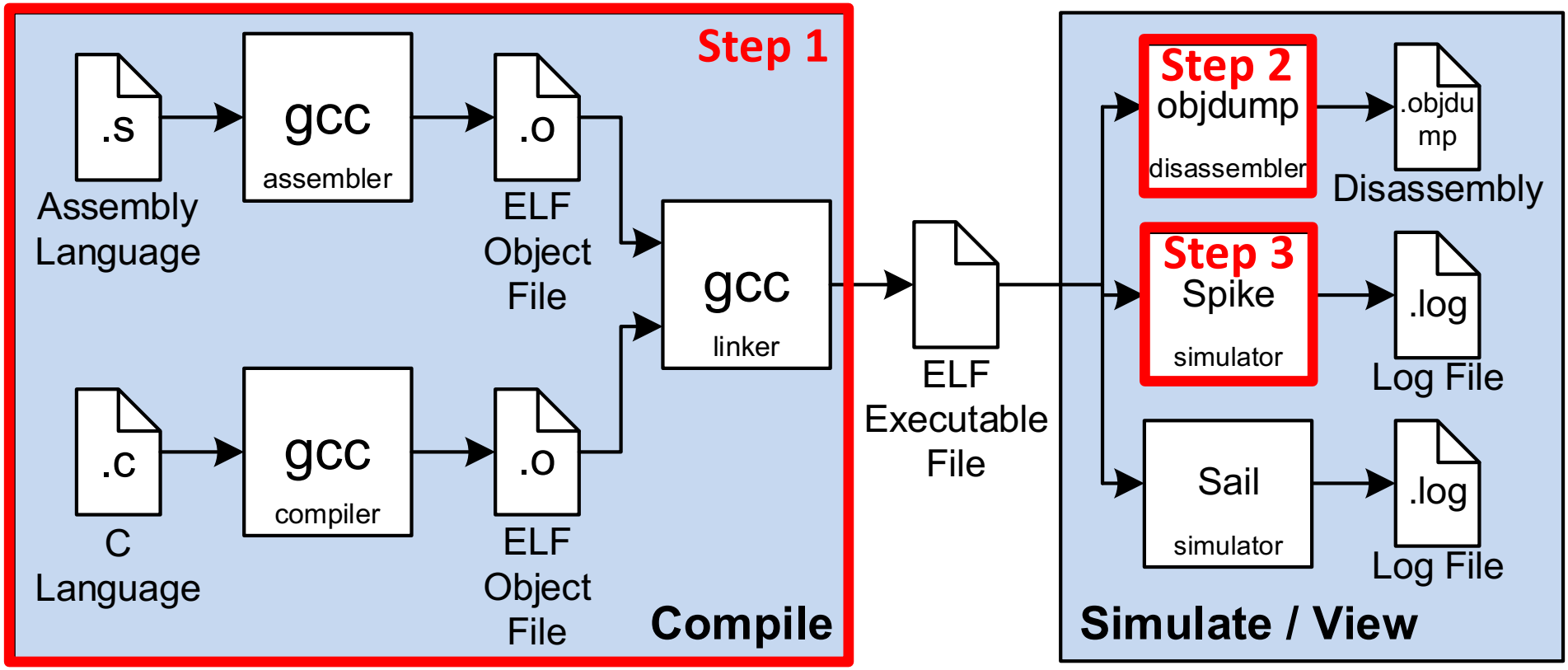
2. **Disassemble** (create human-readable version of) executable file

   ```
   objdump –D <filename>
   ```

3. **Simulate with Spike** (instruction set simulator: ISS)

   ```
   spike --isa=rv32i –d <filename>
   ```

RISC-V Software Tool Flow



Step 1. Assemble

example.S

```
.section .text.init
.globl rvtest_entry_point
rvtest_entry_point:
    li a0, 42

self_loop:
    j self_loop
```

Assemble program (create object file: example.o):

```
$ cd cvw/examples/asm/example
$ riscv64-unknown-elf-gcc -o example -march=rv32i -mabi=ilp32
-nostartfiles -mcmodel=medany -T../../link/link.ld example.S
```

Tool Prefix: `riscv64-unknown-elf-` = cpu-vendor-abi-:

- cpu = riscv64 (but also supports riscv32)
- vendor = unknown (not specific to a flavor of Linux)
- abi = elf (format of binary files: object files / executables)

Step 2. Disassemble

Disassemble object file:

```
$ riscv64-unknown-elf-objdump -D example
```

-D: disassemble contents of all sections, not just code

```
example: file format elf32-littleriscv

Disassembly of section .text:

80000000 <rvtest_entry_point>:
80000000:     02a00513      li a0,42

80000004 <self_loop>:
80000004:     0000006f      j 80000004 <self_loop>

Disassembly of section .riscv.attributes:

00000000 <.riscv.attributes>:
   0:  1941                addi   s2,s2,-16
   2:  0000                unimp
...
```

example.S

```
.section .text.init
.globl rvtest_entry_point
rvtest_entry_point:
    li a0, 42

self_loop:
    j self_loop
```

Step 3. Simulate using Spike

```
$ spike --isa=rv32i -d example
```

-d: interactive debug mode

Press **enter** to advance one instruction at a time

```
warning: tohost and fromhost symbols not in ELF; can't
communicate with target
:
core   0: 0x00001000 (0x00000297) auipc   t0, 0x0
:
core   0: 0x00001004 (0x02028593) addi    a1, t0, 32
:
core   0: 0x00001008 (0xf1402573) csrr    a0, mhartid
:
core   0: 0x0000100c (0x0182a283) lw      t0, 24(t0)
:
core   0: 0x00001010 (0x00028067) jr      t0
:
core   0: 0x80000000 (0x02a00513) li      a0, 42
: reg 0 a0
0x0000002a
:
core   0: 0x80000004 (0x0000006f) j       pc + 0x0
:
: q
```

Expect warning about undefined symbols

Ignore these: Spike inserts some startup code to initialize `a0` and `a1` to some values we don't care about

Spike jumps to user code at 0x80000000

Type **`reg 0 a0`** to display contents of hart 0's a0 register (is 42 = 0x2a, as expected after `li a0, 42`).

Press **`q`** to quit.

Chapter 3: RISC-V Software Tool Flow

More about Spike

Some Useful Spike Commands

Command	Description
`reg 0 a0`	See value of hart 0 register (in this case, `a0`)
`reg 0 mstatus`	See value of hart 0 CSR (in this case, `mstatus`)
`fregs 0 ft0`	See value of hart 0 floating-point (fp) register (in this case, `ft0`)
`fregd 0 ft0`	See value of hart 0 double-precision fp register (in this case, `ft0`)
`mem 80000004`	Read contents of physical memory at address 0x80000004

`until <condition>`	Runs until condition is satisfied. **Examples:**
	`until pc 0 80000004`
	`until reg 0 a0 2a`
	`until mem 0 80000020 1`

`while <condition>`	Runs while condition is satisfied. **Example:** `while mem 0 80000020 deadbeefdeadbeef`
`run`	Runs indefinitely, printing the PC and instruction as it goes
`Ctrl-c`	Pauses simulation and reenter interactive debug mode
`q`	Ends simulation

Spike HTIF (Host Target Interface)

HTIF: defines convention for communication between host (Spike) and target (program)

- **Signal program completion:**
  - By default, Spike ends in infinite loop to avoid executing instructions in invalid memory.
  - Program can declare 8-byte .dword at label called `tohost`

```
tohost:  # write to HTIF
  .dword 0
```

  - **Write 1 to `tohost`:** causes program to terminate successfully

```
la t1, tohost           # Load address of tohost label into t1
li t0, 1                # 1 for success, 3 for failure
sd t0, 0(t1)            # Send success code
```

  - **Write 3 to `tohost`:** causes Spike to print *** FAILED *** and terminate.
- **Write signature memory out to file:**
  - The signature is in .data section delimited by

```
begin_signature:        # Sig. mem.: 2 entries of 4 bytes each
  .fill 2,4,0xdeadbeef  # Initialize sig. mem. with 0xdeadbeef
end_signature:
```

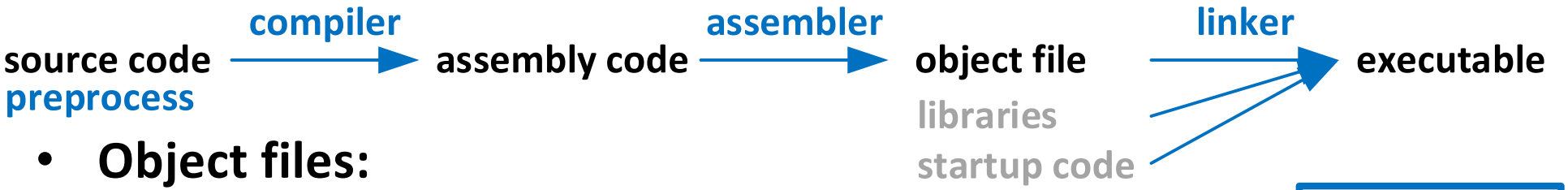
  - **Example:** Writes signature to file: example.signature.output, 4 bytes/line in hex.

```
$ spike +signature=example.signature.output +signature-granularity=4 example
```

Chapter 3: RISC-V Software Tool Flow

Linking & ELF

Compile Process



- **Object files:**
  - Contain machine code
  - Include unresolved external references (e.g., function calls) contained in other object files or libraries
- **Startup code:**
  - Initializes register, including stack and global pointers (`sp` and `gp`)
  - Sets up exception handler
- **Linker:**
  - Creates executable by:
    - Linking only required parts of library. For example, if program only uses printf, it is the only function linked from the library and included in the executable.
    - Placing instructions at consecutive addresses in file.
    - Resolving all references.

Preprocessing inserts #include files and replaces #define macros with their values

ELF: Executable and Linkable Format

ELF files contain:

Object & Executable files are **ELF** files

- **52- or 64-byte header:**
  - Begins with 0x7F454C46 magic number
  - Then has bytes to indicate:
    - 32/64-bit format
    - Little or big endian
    - Code ISA
    - Entry point address where program should begin executing
    - Size and location of program and section header tables
- **Program and section header tables**
- **Sections** (begin with a .)**:**
  - .text: Code
  - .data: Initialized variables
  - .symtab: Symbol table (lists labels and their addresses)
  - And others…

readelf: Print contents of ELF File

```
$ riscv64-unknown-elf-readelf -a example

ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x80000000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          4344 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           40 (bytes)
  Number of section headers:         6
  Section header string table index: 5
```

readelf, cont'd

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        80000000 001000 000008 00  AX  0   0  4
  [ 2] .riscv.attributes RISCV_ATTRIBUTE 00000000 001008 00001a 00      0   0  1
  [ 3] .symtab           SYMTAB          00000000 001024 000070 10      4   5  4
  [ 4] .strtab           STRTAB          00000000 001094 00002e 00      0   0  1
  [ 5] .shstrtab         STRTAB          00000000 0010c2 000033 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x80000000 0x00008 0x00008 R E 0x1000

 Section to Segment mapping:
  Segment Sections...
   00     .text
```

readelf, cont'd

```
Symbol table '.symtab' contains 7 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 80000000     0 SECTION LOCAL  DEFAULT    1 .text
     2: 00000000     0 SECTION LOCAL  DEFAULT    2 .riscv.attributes
     3: 00000000     0 FILE    LOCAL  DEFAULT  ABS cc2lkp4v.o
     4: 80000004     0 NOTYPE  LOCAL  DEFAULT    1 self_loop
     5: 80000000     0 NOTYPE  GLOBAL DEFAULT    1 rvtest_entry_point
     6: 80001000     0 NOTYPE  GLOBAL DEFAULT    1 _end

No version information found in this file.
Attribute Section: riscv
File Attributes
  Tag_RISCV_arch: "rv32i2p0"
```

← Symbol table

Address

Label

← RV32I (rv32i) version 2.0 (2p0)

Chapter 3: RISC-V Software Tool Flow

ABI: Application Binary Interface

ABI: Application Binary Interface

An ABI defines:

- Binary format of executables, object & library files (i.e., ELF)
- **Machine architecture & data types:**
  - Which ISA is supported, called the *machine architecture*
  - Size, layout, and alignment of data types
- **Calling convention** (how arguments are passed)
- **OS system call conventions**
- **Functions for accessing lower-level features**, such as higher-privilege (i.e., S-level) CSR registers
- **Code models** of how instructions access a label in the program

The RISC-V ABI specification is at:
github.com/riscv-non-isa/riscv-elf-psabi-doc

Machine Architectures & Data Types

Machine architectures (ABI-defined RISC-V flavor):

- May be supported by hardware
- And targeted by compiler
- Associated with ABI that specifies data type size.

Machine Architecture	Synonym	ABI
rv32e		ilp32e
rv32i		ilp32
rv32im		ilp32
rv32iac		ilp32
rv32imac		ilp32
rv32imafc		ilp32f
rv32imafdc	rv32gc	ilp32d
rv64i		lp64
rv64im		lp64
rv64ic		lp64
rv64iac		lp64
rv64imac		lp64
rv64imafdc	rv64gc	lp64d

i = integer
l = long
p = pointer

i.e., ilp32: int, long, pointer = 32 bits

Data Type	ilp32	lp64
`int`	32	32
`long`	32	64
`pointers`	32	64
`char`	8	8
`short`	16	16

Calling Convention

RISC-V Base Integer ABIs (ilp32/lp64):

- `a0–a7`: Argument registers
- `a0` (and `a1` if result is 2XLEN wide): Return value
- `s0–s11`: Preserved across function call
- `t0–t6`: Not preserved across function call

RISC-V Floating-Point ABIs (ilp64f/d and lp64d):

- `fa0–fa7`: Argument registers
- `fs0–fs11`: Preserved across function call
- `ft0–ft6`: Not preserved across function call

System Calls

When U-mode program calls OS (S-mode):

- Place syscall number in `a7`
- Other arguments in `a0–a5`
- Execute `ecall` (executive call) instruction:
  - Causes trap
  - Elevates privilege to S-mode
  - Trap handler in OS kernel:
    - Looks at `a7` and arguments to perform appropriate system call
    - Places return result in `a0`
    - Performs `sret` to return to U-mode program

Example system call (from U-mode to S-mode):

```
int open(const char *pathname, int flags);
```

- `a0`: *pathname* = pointer to file path name
- `a1`: *flags* (indicate whether need to create new file, etc.)
- `a0`: returns file descriptor = index into OS's table of open files.

Code Models

Small (medlow):

- 32-bit absolute addresses
- Thus, can access variables & code in lower 2 GiB of address space
- Not position independent, so linker must change addresses when relocating code.

Medium (medany):

- Uses `auipc` to add signed 32-bit offset to PC
- Thus, can access variables & code within +/- 2 GiB of the current instruction
- Position independent, so linker must **not** change addresses when relocating code.

Chapter 3: RISC-V Software Tool Flow

SBI: Supervisor Binary Interface

SBI: Supervisor Binary Interface

OS invokes platform runtime firmware using the SBI:

- **OS makes this call by:**
  - Placing SBI call information in a6 and a7
  - Placing any other arguments in other a registers
  - Executes ecall, which calls trap handler and elevates privilege from S-mode to M-mode
  - The trap handler in SBI platform runtime firmware:
    - Does the necessary M-mode work
    - Places error code, if any, in a0
    - Places return result, if any, in a1
    - Performs mret to return to OS

Example system call (from S-mode to M-mode):

```
struct sbiret sbi_set_timer(uint64_t stime_value);
```

- Sets current value of timer
- a7: EID (extension ID) = 54494D45
- a6: FID (function ID) = 54494D45
- a0: *stime_value* (new time)

SBI System Call Example

Example system call (from S-mode to M-mode):

```
struct sbiret sbi_set_timer(uint64_t stime_value);
```

- Sets current value of timer
- a7: EID (extension ID) = 54494D45
- a6: FID (function ID) = 54494D45
- a0: *stime_value* (new time)

OS invokes platform runtime firmware using the SBI:

- The timer is part of a memory-mapped peripheral (such as the Core-Local Interruptor: CLINT)
  - Address may be system-dependent
  - Can only be accessed in machine mode
- Thus, the SBI provides this platform-independent call so OS can access the timer with standard code.

Other SBI System Calls

Examples of other SBI system calls include:

- Set timer
- Send inter-processor interrupt (IPI, also called software interrupt) to another hart
- Remote fences to synchronize with other harts
- Start or stop harts
- Reboot system
- Read performance counters
- Check versions of the hardware and SBI implementation

See the SBI documentation for a list of SBI calls:
https://github.com/riscv-non-isa/riscv-sbi-doc

Chapter 3: RISC-V Software Tool Flow

Assembler Directives

Assembler Directives

Assembler Directive	Description
Sections	
`.text`	Text (program code)
`.data`	Data: initialized variables
`.rodata`	Read-only data: initialized constants
`.bss`	Data: uninitialized variables (usually initialized to 0)
`.symtab`	Symbol table, including addresses of labels
`.section name`	User-defined section
`.end`	Optional end of file
Data	
`.byte x1, x2, …`	8-bit comma-separated values
`.half/.short/.2byte x1, x2, …`	16-bit comma-separated values
`.word/.long/.4byte x1, x2, …`	32-bit comma-separated values
`.dword/.8byte x1, x2, …`	64-bit comma-separated values
`.string “str”`	Null terminated string
`.space size`	Leave room for size bytes
`.fill num, size, value`	Fill memory with num entries of size bytes with value
Other	
`.align size`	Align next code or data address to a power of 2 size
`.global/.globl`	Define a global symbol visible to the linker
`.equ const, value`	Define a constant const to be value
`.macro … .endm`	Define a macro

Chapter 3: RISC-V Software Tool Flow

Performance Measurement

Performance Measurement CSRs

Most common performance measurement CSRs:

- **`instret`**
  - Number of instructions retired (completed)
- **`cycle`**
  - Number of processor clock cycles that have passed
- Access these using **CSR read** (`csrr`) pseudoinstruction:

  Examples:

```
csrr t1, instret
csrr t2, cycle
```

Note: in an ISS such as Spike (i.e., that models only function of instructions, not hardware – so CPI = 1), `instret` = `cycle`.

Chapter 3: RISC-V Software Tool Flow

Putting It All Together

Example Showing Tool Flow

Program (`sumtest.S`):

- Links multiple files (`sumtest.S` and `sum.S`)
- Demonstrates use of assembler directives
- Calls function `sum(N)` in `sum.S`
- Writes return result and instruction count to signature memory
- Writes 1 to host interface (`tohost` label) to indicate completion before entering infinite loop at label `self_loop`.

Example Program: sumtest.s

```
// sumtest.S
.global rvtest_entry_point
rvtest_entry_point:
    la sp, topofstack       # Initialize stack pointer
    la t0, N                # Get address of N in data
    ld a0, 0(t0)            # Load N
    csrr s8, instret        # Count # instructions completed before call made
    jal sum                 # Call sum(N)
    csrr s9, instret        # Count # instructions completed after call made
    sub s9, s9, s8          # Calculate # instructions completed during call
    la t0, begin_signature  # Address of signature
    sd a0, 0(t0)            # Store sum(N) in signature
    sd s9, 8(t0)            # Record performance (# instr. executed during sum)

write_tohost:
    la t1, tohost           # Load address of tohost label into t1
    li t0, 1                # 1 for success, 3 for failure
    sd t0, 0(t1)            # Send success code

self_loop:
    j self_loop             # Wait
```

Example Program: sumtest.s

continued…

```
.section .tohost
tohost:                             # Write to HTIF
  .dword 0
fromhost:
  .dword 0

.data
N:
  .dword 4

.EQU XLEN,64
begin_signature:
  .fill 2*(XLEN/32),4,0xdeadbeef # Initialize signature memory with 0xdeadbeef
end_signature:                      # 2 XLEN words, each 4 bytes = 0xdeadbeef

# Initialize stack with room for 512 bytes
.bss
    .space 512
topofstack:
```

Second file: sum.s

```
// sum.S
// Add numbers from 1 to N.
// result in s0, i in s1, N in a0, return value in a0

.global sum
sum:
     addi sp, sp, -16     # Make room on the stack
     sd s0, 0(sp)
     sd s1, 8(sp)

     li s0, 0             # result = 0
     li s1, 1             # i = 1
for: bgt s1, a0, done     # Exit loop if i > n
     add s0, s0, s1       # result = result + i
     addi s1, s1, 1       # i++
     j for                # Repeat

done:
     mv a0, s0            # Put result in a0 to return
     ld s0, 0(sp)         # Restore s0, s1 from stack
     ld s1, 8(sp)
     addi sp, sp, 16
     ret                  # Return from function
```

```
// C Equivalent

long sum(long N) {
  long result, i;

  result = 0;
  for (i=1; i<=N; i++)
    result = result + i;
  return result;
}
```

Results: Register Values

```
// sum.S
// Add numbers from 1 to N.
// result in s0, i in s1, N in a0, return value in a0

.global sum
sum:
     addi sp, sp, -16     # Make room on the stack
     sd s0, 0(sp)
     sd s1, 8(sp)

     li s0, 0             # result = 0
     li s1, 1             # i = 1
for: bgt s1, a0, done     # Exit loop if i > n
     add s0, s0, s1       # result = result + i
     addi s1, s1, 1       # i++
     j for                # Repeat

done:
     mv a0, s0            # Put result in a0 to return
     ld s0, 0(sp)         # Restore s0, s1 from stack
     ld s1, 8(sp)
     addi sp, sp, 16
     ret                  # Return from function
```

Call to `sum` takes **13 + 4N** instructions:

- **1** for `jal` to call `sum`
- **10** before and after for loop
- **4** for each iteration of loop (thus **4N**)
- **1** for last iteration of loop (when `bgt` is not taken)
- **1** for `csrr` (after `sum` returns) to read `instret`

Thus, for `sum(4)`, we expect:

- Return value: `a0` = 1+2+3+4 = 10 = **0xA**
- `instret` = 13 + 4*4 = 29 = **0x1D**

Results: Signature Memory

```
// sumtest.S
.global rvtest_entry_point
rvtest_entry_point:
    la sp, topofstack       # Initialize stack pointer
    la t0, N                # Get address of N in data
    ld a0, 0(t0)            # Load N
    csrr s8, instret        # Count # instructions completed before call made
    jal sum                 # Call sum(N)
    csrr s9, instret        # Count # instructions completed after call made
    sub s9, s9, s8          # Calculate # instructions completed during call
    la t0, begin_signature  # Address of signature
    sd a0, 0(t0)            # Store sum(N) in signature
    sd s9, 8(t0)            # Record performance (# instr. executed during sum)
```

For `sum(4)`:

- Return value: `a0` = 1+2+3+4 = 10 = **0xA**
- `instret` = 13 + 4*4 = 29 = **0x1D**

Signature Memory (written out to file: sumtest.signature.output)

```
000000000000000A
000000000000001D
```

Makefile for Assembly Programs

```
TARGET = sumtest

$(TARGET).objdump: $(TARGET)
  riscv64-unknown-elf-objdump -D $(TARGET) > $(TARGET).objdump

$(TARGET): $(TARGET).S sum.S
  riscv64-unknown-elf-gcc -g -o $(TARGET) -march=rv64gc -mabi=lp64d -mcmodel=medany \
        -nostartfiles -T../../link/link.ld $(TARGET).S sum.S

sim:
  spike +signature=$(TARGET).signature.output +signature-granularity=8 $(TARGET)
  diff --ignore-case $(TARGET).signature.output $(TARGET).reference_output || exit
  echo "Signature matches! Success!"

clean:
rm -f $(TARGET) $(TARGET).objdump $(TARGET).signature.output
```

You can easily modify the Makefile to target a new program by changing these two lines:

Change name of program:

```
TARGET = sumtest
```

Change name of dependency/dependencies:

```
$(TARGET): $(TARGET).S sum.S
```

Running Makefile

```
$ make clean    # optionally delete old files

$ make          # build and disassemble sumtest

$ make sim      # simulate sumtest in spike
                # diff output with reference output:
                # diff --ignore-case sumtest.signature.output
                # sumtest.reference_output || exit
                # echo "Signature matches! Success!"
```

Result:

```
Signature matches! Success!
```

sumtest.objdump

```
sumtest:     file format elf64-littleriscv

Disassembly of section .text:
0000000080000000 <rvtest_entry_point>:
    80000000:    00003117   auipc sp,0x3
    80000004:    20010113   addi  sp,sp,512 # 80003200 <_end>
    80000008:    00002297   auipc t0,0x2
    8000000c:    ff828293   addi  t0,t0,-8 # 80002000 <N>
    80000010:    0002b503   ld    a0,0(t0)
    80000014:    c0202c73   rdinstret      s8
    80000018:    02c000ef   jal   ra,80000044 <sum>
    8000001c:    c0202cf3   rdinstret      s9
    80000020:    418c8cb3   sub   s9,s9,s8
    80000024:    00002297   auipc t0,0x2
    80000028:    fe428293   addi  t0,t0,-28 # 80002008 <begin_signature>
    8000002c:    00a2b023   sd    a0,0(t0)
    80000030:    0192b423   sd    s9,8(t0)
0000000080000034 <write_tohost>:
    80000034:    00001317   auipc t1,0x1
    80000038:    fcc30313   addi  t1,t1,-52 # 80001000 <tohost>
    8000003c:    4285       li    t0,1
    8000003e:    00533023   sd    t0,0(t1)
0000000080000042 <self_loop>:
    80000042:    a001       j     80000042 <self_loop>
```

sumtest.objdump, cont'd

```
0000000080000044 <sum>:
    80000044:      1141          addi   sp,sp,-16
    80000046:      e022          sd     s0,0(sp)
    80000048:      e426          sd     s1,8(sp)
    8000004a:      4401          li     s0,0
    8000004c:      4485          li     s1,1
000000008000004e <for>:
    8000004e:      00954563      blt    a0,s1,80000058 <done>
    80000052:      9426          add    s0,s0,s1
    80000054:      0485          addi   s1,s1,1
    80000056:      bfe5          j      8000004e <for>
0000000080000058 <done>:
    80000058:      8522          mv     a0,s0
    8000005a:      6402          ld     s0,0(sp)
    8000005c:      64a2          ld     s1,8(sp)
    8000005e:      8082          ret
Disassembly of section .tohost:
0000000080001000 <tohost>:
Disassembly of section .data:
0000000080002000 <N>:
    80002000:      0004                         0x4
0000000080002008 <begin_signature>:
    80002008:      deadbeef
    8000200c:      deadbeef
    80002010:      deadbeef
    80002014:      deadbeef
```

sumtest ELF Metadata

```
$ riscv64-unknown-elf-readelf -a sumtest

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  ...
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000080000000  00001000
       0000000000000060  0000000000000000  AX       0     0     2
  [ 2] .tohost           PROGBITS         0000000080001000  00002018
       0000000000000010  0000000000000000           0     0     1
  [ 3] .data             PROGBITS         0000000080002000  00002000
       0000000000000018  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000080003000  00002018
       0000000000000200  0000000000000000  WA       0     0     1
  [ 5] .riscv.attributes RISCV_ATTRIBUTE  0000000000000000  00002028
       0000000000000037  0000000000000000           0     0     1
  ...
```

sumtest ELF Metadata

continued…

```
...
Symbol table '.symtab' contains 28 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000080000000     0 SECTION LOCAL  DEFAULT    1 .text
     2: 0000000080001000     0 SECTION LOCAL  DEFAULT    2 .tohost
     3: 0000000080002000     0 SECTION LOCAL  DEFAULT    3 .data
     4: 0000000080003000     0 SECTION LOCAL  DEFAULT    4 .bss
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5 .riscv.attributes
    ...
    13: 0000000080003200     0 NOTYPE  LOCAL  DEFAULT    4 topofstack
    14: 0000000080002000     0 NOTYPE  LOCAL  DEFAULT    3 N
    15: 0000000080002008     0 NOTYPE  LOCAL  DEFAULT    3 begin_signature
    16: 0000000080000034     0 NOTYPE  LOCAL  DEFAULT    1 write_tohost
    17: 0000000080001000     0 NOTYPE  LOCAL  DEFAULT    2 tohost
    18: 0000000080000042     0 NOTYPE  LOCAL  DEFAULT    1 self_loop
    19: 0000000080001008     0 NOTYPE  LOCAL  DEFAULT    2 fromhost
    20: 0000000000000040     0 NOTYPE  LOCAL  DEFAULT  ABS XLEN
    21: 0000000080002018     0 NOTYPE  LOCAL  DEFAULT    3 end_signature
    22: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS ccZtnxq7.o
    23: 000000008000004e     0 NOTYPE  LOCAL  DEFAULT    1 for
    24: 0000000080000058     0 NOTYPE  LOCAL  DEFAULT    1 done
    25: 0000000080000044     0 NOTYPE  GLOBAL DEFAULT    1 sum
    26: 0000000080000000     0 NOTYPE  GLOBAL DEFAULT    1 rvtest_entry_point
    27: 0000000080003200     0 NOTYPE  GLOBAL DEFAULT    4 _end
```

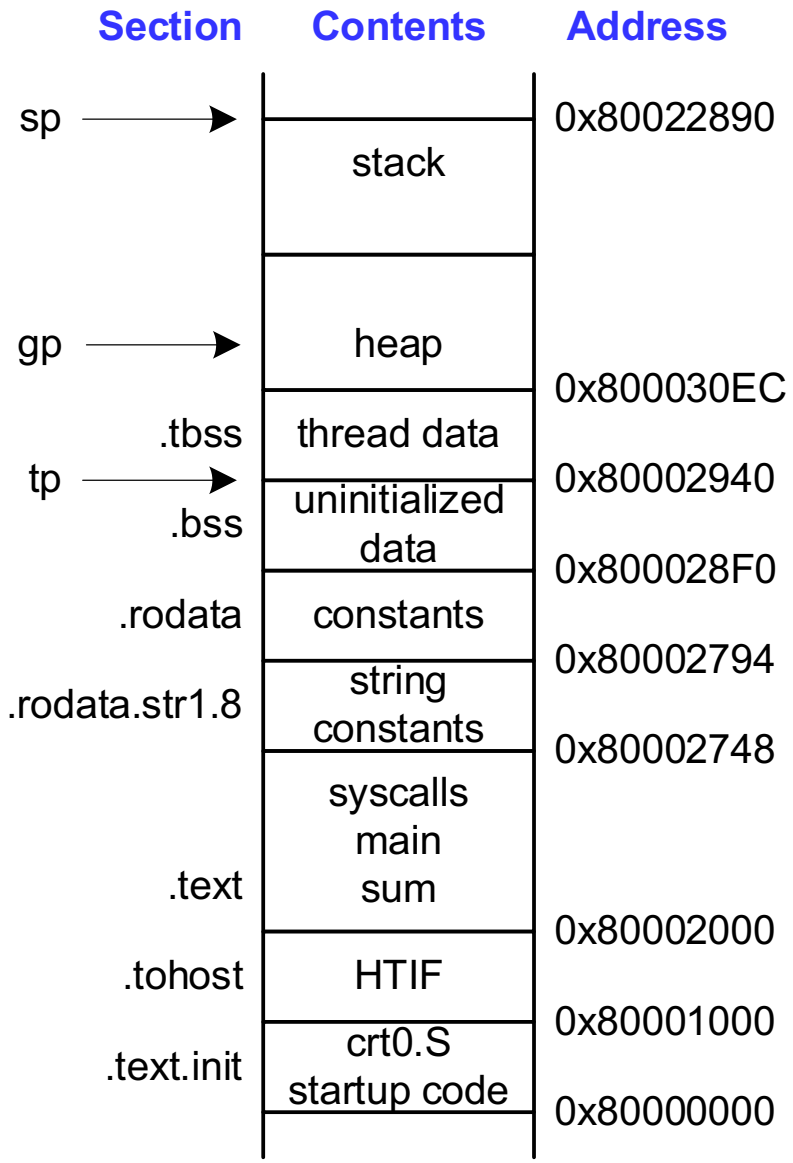
Chapter 3: RISC-V Software Tool Flow

GCC's GNU C Compiler

Simple C Program

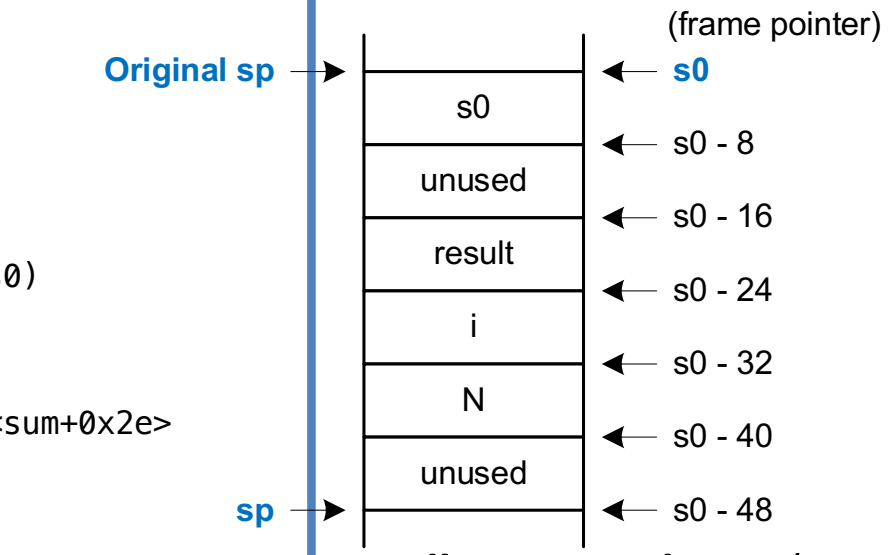
```
// sum.c
#include <stdio.h>  // supports printf
#include "util.h"   // supports verify

long sum(long N) {
  long result, i;
   result = 0;
   for (i=1; i<=N; i++) {
       result = result + i;
   }
   return result;
}

int main(void) {
    int s[1], expected[1];
    setStats(1);
    s[0] = sum(4);
    setStats(0);
    printf("s = %d\n", s[0]);
    expected[0] = 10;
    return verify(1, s, expected); // 0 means success
}
```

Program Memory Map



Disassembly: No Optimization

```
long sum(long N) {
    8000231a:  7179                addi   sp,sp,-48
    8000231c:  f422                sd     s0,40(sp)
    8000231e:  1800                addi   s0,sp,48
    80002320:  fca43c23            sd     a0,-40(s0)
  long result, i;
   result = 0;
    80002324:  fe043423            sd     zero,-24(s0)
   for (i=1; i<=N; i++) {
    80002328:  4785                li     a5,1
    8000232a:  fef43023            sd     a5,-32(s0)
    8000232e:  a829                j      80002348 <sum+0x2e>
       result = result + i;
    80002330:  fe843703            ld     a4,-24(s0)
    80002334:  fe043783            ld     a5,-32(s0)
    80002338:  97ba                add    a5,a5,a4
    8000233a:  fef43423            sd     a5,-24(s0)
   for (i=1; i<=N; i++) {
    8000233e:  fe043783            ld     a5,-32(s0)
    80002342:  0785                addi   a5,a5,1
    80002344:  fef43023            sd     a5,-32(s0)
    80002348:  fe043703            ld     a4,-32(s0)
    8000234c:  fd843783            ld     a5,-40(s0)
    80002350:  fee7d0e3            bge    a5,a4,80002330 <sum+0x16>
   }
   return result;
    80002354:  fe843783            ld     a5,-24(s0)
}
    80002358:  853e                mv     a0,a5
    8000235a:  7422                ld     s0,40(sp)
    8000235c:  6145                addi   sp,sp,48
    8000235e:  8082                ret
```



- **Allocates 48 bytes** (6 double words) on stack.
- **Saves s0** on stack & uses it as frame pointer (points to bottom of stack frame).
- Keeps **local variables on stack** instead of registers.
- Grossly **inefficient** code.

Disassembly: -O

Basic optimizations

```
0000000080002000 <sum>:
#include "util.h"   // supports verify

long sum(long N) {
  long result, i;
   result = 0;
   for (i=1; i<=N; i++) {
    80002000:  00a05b63              blez   a0,80002016 <sum+0x16>
    80002004:  00150713              addi   a4,a0,1
    80002008:  4785                  li     a5,1
   result = 0;
    8000200a:  4501                  li     a0,0
      result = result + i;
    8000200c:  953e                  add    a0,a0,a5
   for (i=1; i<=N; i++) {
    8000200e:  0785                  addi   a5,a5,1
    80002010:  fee79ee3              bne    a5,a4,8000200c <sum+0xc>
    80002014:  8082                  ret
   result = 0;
    80002016:  4501                  li     a0,0
   }
   return result;
}
    80002018:  8082                  ret
```

- **No stack** usage
- **Register Mapping:**
  - N in `a0`
  - N+1 in `a4` (completion condition)
  - `i` in `a5`
  - `result` in `a0`
- Disassembler has some **difficulty associating assembly code with original C code** because optimizer reordered some of the code for performance.

Disassembly: -O2 or –O3

```
8000271a:     45a9                    li          a1,10
```

- Compiler **precomputes result** of calling `sum(4)` and replaces sum function call with assignment to 10.
- The **call to `verify` is also eliminated** because the compiler precomputes that it will return 0 for success.

Performance

Optimization Level	`mcycle`	`minstret`	True Cycles = True # Instructions
None	115	132	62
-O	31	38	21
-O2 or -O3	11	16	1

- In Spike, **CPI = 1**
- So, **ideally all columns should be the same**.
- But in Spike there is **overhead to call setStats**, which is an underlying function to read each performance counter.
- These **performance counters are more useful for longer programs**, where overhead is negligible.

Chapter 3: RISC-V Software Tool Flow

Mixing C and Assembly Code

C Code with Inline Assembly

```
// inline.c
#include <stdio.h>

int main(void) {
  long cycles;   // held in a1

  // read mcycle register: csrr a1, mcycle
  asm volatile("csrr %0, 0xB00" : "=r"(cycles));
  printf ("mcycle = %ld\n", cycles);
}
```

C code with inline assembly

```
int main(void) {
    80002000:  1141     addi   sp,sp,-16
    80002002:  e406     sd     ra,8(sp)
    long cycles;
    asm volatile("csrr %0, 0xB00" :
"=r"(cycles));
    80002004:  b00025f3 csrr   a1,mcycle
    printf ("mcycle = %ld\n", cycles);
    80002008:  00000517 auipc  a0,0x0
    8000200c:  70850513 addi   a0,a0,1800 #
80002710 <atol+0x6a>
    80002010:  540000ef jal    ra,80002550
<printf>
 }
    80002014:  4501     li     a0,0
    80002016:  60a2     ld     ra,8(sp)
    80002018:  0141     addi   sp,sp,16
    8000201a:  8082     ret
```

Disassembly of C code

Program with C and Assembly Files

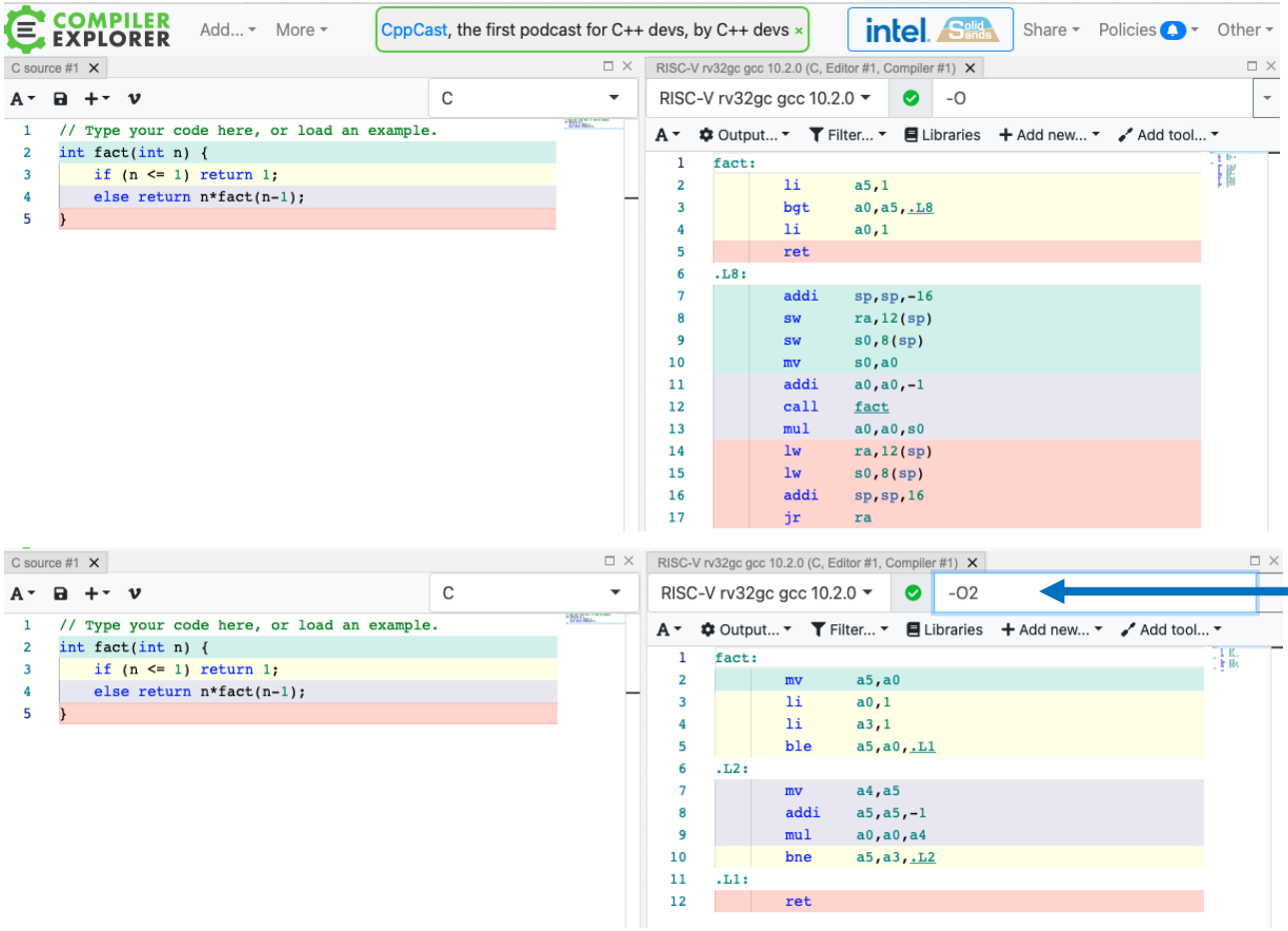
```c
// sum_mixed.c
// Call assembly language function
// from C function

#include <stdio.h>
#include "util.h"
extern int sum(int);

int main(void) {
    int s[1], expected[1];

    setStats(1);
    s[0] = sum(4);
    setStats(0);
    printf("s = %d\n", s[0]);
    expected[0] = 10;
    // return 0 means success
    return verify(1, s, expected);
}
```

```
// sum.S – same as before, in earlier slides
// Add numbers from 1 to N.
// result in s0, i in s1, N in a0, return value in a0

.global sum
sum:
     addi sp, sp, -16    # Make room on the stack
     sd s0, 0(sp)
     sd s1, 8(sp)

     li s0, 0            # result = 0
     li s1, 1            # i = 1
for: bgt s1, a0, done    # Exit loop if i > n
     add s0, s0, s1      # result = result + i
     addi s1, s1, 1      # i++
     j for               # Repeat

done:
     mv a0, s0           # Put result in a0 to return
     ld s0, 0(sp)        # Restore s0, s1 from stack
     ld s1, 8(sp)
     addi sp, sp, 16
     ret                 # Return from function
```

- In assembly file, list sum as **.global**:
  `.global sum`
- In C file, list sum as **extern**:
  `extern int sum(int);`

Quick Tool: Compiler Explorer

```
// Recursive factorial function: return n*(n-1)*...
int fact(int n) {
  if (n <= i) return 1;
  else return n*fact(n-1);
```



Compiler Explorer available at **Godbolt.org**

- Choose architecture and compiler (e.g., rv32gc, gcc 10.2.0)
- Optionally choose compiler flags
- Type C code into left pane.
- Disassembled output appears in right pane.

-O2 is clever enough to **eliminate recursive** function call.

Chapter 3: RISC-V Software Tool Flow

QEMU Simulation

QEMU: Open-Source Emulator

QEMU: powerful open-source emulator that handles many different architectures, including RISC-V

- Performs **just-in-time translation** from the target architecture being simulated (e.g., RISC-V) to the architecture of the host computer (e.g., x86)
- Runs **~1 billion instructions/second** – can comfortably boot an operating system in emulation (e.g., Linux)
- An **order of magnitude faster than Spike**
- Interfaces with GDB
- Emulates **peripherals important for booting an OS** on the target architecture

Simulating with QEMU

```
$ cd riscv-wally/examples/asm/sumtest
$ qemu-system-riscv64 -S -M virt -nographic -bios none -kernel sumtest
Ctrl-a c
QEMU 6.0.91 monitor - type 'help' for more information
(qemu) info registers
 pc       0000000000001000
… <other registers>
(qemu) x /2xg 0x80002008
0000000080002008: 0xdeadbeefdeadbeef
(qemu) c
(qemu) info registers
 pc       0000000080000042
… <other registers>
(qemu) x /2xg 0x80002008
0000000080002008: 0x000000000000000a 0x0000000000048470
(qemu) q
```

Flags:

-S	Starts with QEMU paused at beginning of program
-M virt	Use generic virtual RISC-V platform instead of specific board
-nographic	Disables GUI support – instead redirects to display on console
-bios none	Skips loading OpenSBI BIOS and runs in bare metal mode
-kernel sumtest	Loads sumtest into program memory

Simulating with QEMU

```
$ cd riscv-wally/examples/asm/sumtest
$ qemu-system-riscv64 -S -M virt -nographic -bios none -kernel sumtest
Ctrl-a c
QEMU 6.0.91 monitor - type 'help' for more information
(qemu) info registers
 pc       0000000000001000
… <other registers>
(qemu) x /2xg 0x80002008
0000000080002008: 0xdeadbeefdeadbeef
(qemu) c
(qemu) info registers
 pc       0000000080000042
… <other registers>
(qemu) x /2xg 0x80002008
0000000080002008: 0x000000000000000a 0x0000000000048470
(qemu) q
```

At end of program `instret` has wildly inaccurate number (0x48470). QEMU depends on host's performance counters, which run at different rate than emulated processor.

Commands:

Ctrl-a c	Enters monitor
info registers	Prints all registers (i.e., PC = 0x1000, where QEMU runs small bootloader)
/x /2xg 0x80002008	Print signature memory
c	Continue program – will automatically halt at infinte loop
q	Exit QEMU

Other QEMU Features

To get accurate cycle count

- Enable virtual instruction counter (at the cost of simulation speed):

  `-icount 0`

- Can also omit –S to start immediately.

```
$ qemu-system-riscv64 -M virt -nographic -bios none -kernel sumtest -icount 0
Ctrl-a c
QEMU 6.0.91 monitor - type 'help' for more information
(qemu) x /2xg 0x80002008
0000000080002008: 0x000000000000000a 0x000000000000001d
(qemu) q
```

Now `instret` is correct: 13 + 4N = 13 + 4*4 = 29 = 0x1d

Debugging with GDB

How to interface QEMU to GDB debugger:

```
$ qemu-system-riscv64 -S -M virt -nographic -bios none -kernel sumtest -gdb
tcp::1234
```

Then invoke gdb in another terminal, connect it over the same port, and load the symbols from `sumtest`.

The authoritative book on GDB is freely available at:
sourceware.org/gdb/current/onlinedocs/gdb.pdf

Common GDB Commands

GDB Command	Description
`b main`	Set breakpoint at function (in this case, `main`)
`info break`	List breakpoints
`d N`	Delete breakpoint number N
`s`	Single-step one statement
`p N`	Print the value of a variable (in this case, N)
press return/enter	Repeats last command
n	Execute entire function (like step over)
Ctrl-c	Pause simulation
q	Quit GDB

Example GDB Transcript

```
$ cd $WALLY/examples/C/sum
$ qemu-system-riscv64 -M virt -nographic -bios none -kernel sum -S -gdb tcp::1234
in another terminal:
$ riscv64-unknown-elf-gdb -iex "file sum" -ex "target remote:1234"
GNU gdb (GDB) 10.1
…
Reading symbols from sum...
Remote debugging using :1234
0x0000000000001000 in ?? ()
(gdb) b main
Breakpoint 1 at 0x8000201a: main. (2 locations)
(gdb) c
Continuing.

Breakpoint 1, main () at sum.c:17
17          int main(void) {
(gdb) s
19              setStats(1);
(gdb) n
20              s[0] = sum(4);
(gdb) s
sum (N=N@entry=4) at sum.c:11
11              for (i=1; i<=N; i++) {
(gdb) p N
$1 = 4
(gdb) s
12                  result = result + i;
(gdb) p i
$2 = 1
(gdb) s
11              for (i=1; i<=N; i++) {
(gdb) p result
$3 = 1
(gdb) s
12                  result = result + i;
(gdb)
11              for (i=1; i<=N; i++) {
(gdb)
12                  result = result + i;
(gdb)
11              for (i=1; i<=N; i++) {
(gdb)
12                  result = result + i;
(gdb)
11              for (i=1; i<=N; i++) {
(gdb)
main () at sum.c:21
21              setStats(0);
(gdb) p s[0]
$4 = <optimized out>
(gdb)
$5 = <optimized out>
(gdb) n
22              printf("s = %d\n", s[0]);
(gdb) n
^C
Program received signal SIGINT, Interrupt.
0x00000000800020c2 in syscall (arg2=7,
arg1=<optimized out>, arg0=1, which=64)
    at ../common/syscalls.c:28
28          while (fromhost == 0)
(gdb) q
```

Chapter 3: RISC-V Software Tool Flow

RISC-V Test Suites

Test Suites

- The RISC-V Foundation has Special Interest Groups (SIGs) to guide aspects of the open-source architecture.
- RISC-V Foundation's **Architecture Test SIG** provides test suite for unprivileged instructions:

  riscv-arch-test
- Test suites and verification methods are rapidly evolving, but in the interim, we have created additional test, especially of privileged instructions, in:

  wally-riscv-arch-test
- To compile all the test cases, go to top-level `CVW` directory, type:

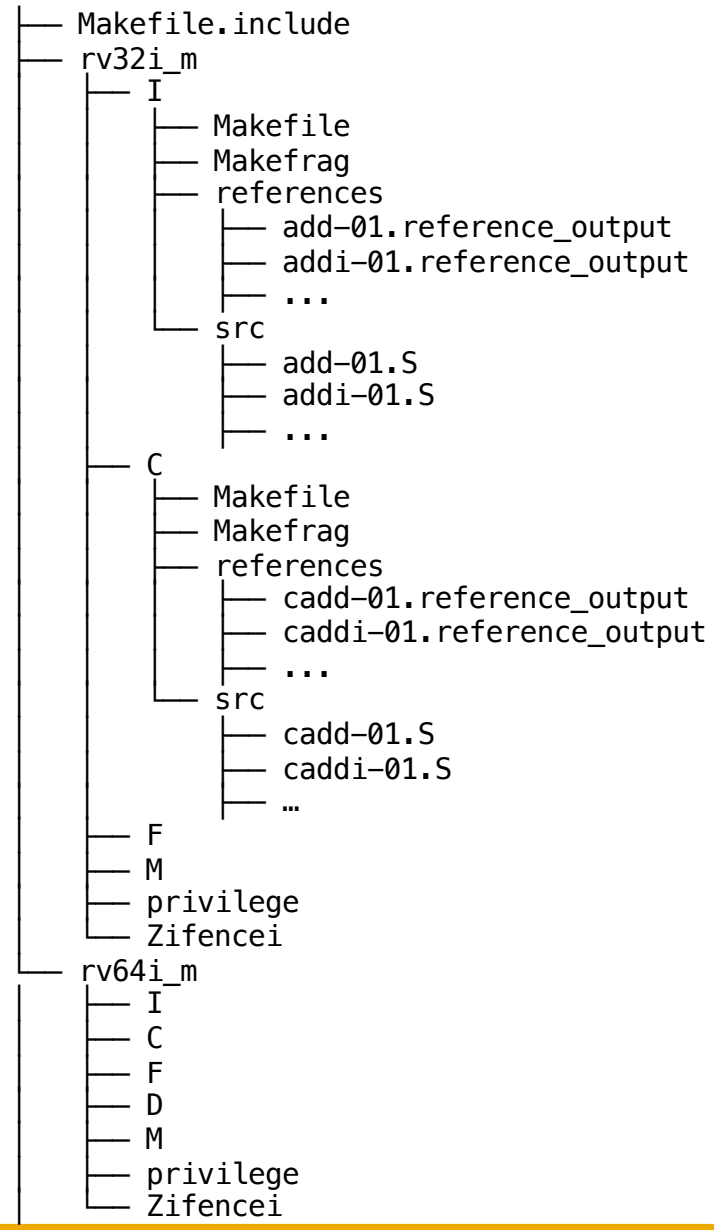
```
$ make regression
```

riscv-arch-test

- **Provides test cases for:**
  - RV32I, RV64I and C, M, F, and D extensions
  - Basic privileged instruction tests.
- **Each instruction has assembly language test file:**
  - Exercises the instruction
  - Generates a `signature.output` file
  - Compares that file with `reference_output` file containing the expected results.
- An intricate system of Makefiles compiles the tests with GCC, runs them with Spike, and verifies that the signature matches the reference.
- `riscv-arch-test` is incorporated into the `cvw` repository as a git submodule so that improvements in the suite can easily be pulled into `cvw`.

Tests located in:
`cvw/addins/riscv-arch-test/riscv-test-suite`

Hierarchy of riscv-test-suites



- Each **extension** has its own folder.
- Within an extension, each **instruction** has its own assembly file.

Example: add-01.S

```
#include "model_test.h"
#include "arch_test.h"
RVTEST_ISA("RV32I")                 # Does nothing

.section .text.init
.globl rvtest_entry_point
rvtest_entry_point:
RVMODEL_BOOT                        # Does nothing
RVTEST_CODE_BEGIN                   # Initializes registers to predictable values
# 80000000:         feedc0b7   lui      ra,0xfeedc
# 80000004:         ead08093   addi     ra,ra,-339 # feedbead <_end+0x7eed63a9>
# 80000008:         ff76e137   lui      sp,0xff76e
# 8000000c:         f5610113   addi     sp,sp,-170 # ff76df56 <_end+0x7f768452>
...
RVTEST_SIGBASE( x3,signature_x3_1)   # Sets x3 to the signature section of the data segment
# 800000f8:         00005197   auipc    gp,0x5
# 800000fc:         f1818193   addi     gp,gp,-232 # 80005010 <begin_signature>

inst_0: # Test add x24, x4, x24 with x4 = 0x7fffffff, x24 = 0x00000001
// rs2 == rd != rs1, rs1==x4, rs2==x24, rd==x24, rs1_val > 0 and rs2_val > 0, rs2_val == 1,
// rs1_val == (2**(xlen-1)-1), rs1_val != rs2_val, rs1_val == 2147483647
// opcode: add ; op1:x4; op2:x24; dest:x24; op1val:0x7fffffff;  op2val:0x1
TEST_RR_OP(add, x24, x4, x24, 0x80000000, 0x7fffffff, 0x1, x3, 0, x18)
# 80000100:         80000237   lui      tp,0x80000
# 80000104:         fff20213   addi     tp,tp,-1 # 7fffffff <_end+0xffffa4fb>
# 80000108:         00100c13   li       s8,1
# 8000010c:         01820c33   add      s8,tp,s8
# 80000110:         0181a023   sw       s8,0(gp)
```

Example: add-01.S, cont'd

```
inst_1: # Test add x28, x10, x10 with x10 = 0x00020000
// rs1 == rs2 != rd, rs1==x10, rs2==x10, rd==x28, rs1_val > 0 and rs2_val < 0, rs2_val == -
257,
// rs1_val == 131072
// opcode: add ; op1:x10; op2:x10; dest:x28; op1val:0x20000;  op2val:0x20000
TEST_RR_OP(add, x28, x10, x10, 0x40000, 0x20000, 0x20000, x3, 4, x18)
# 80000114:          00020537   lui      a0,0x20
# 80000118:          00020537   lui      a0,0x20
# 8000011c:          00a50e33   add      t3,a0,a0
# 80000120:          01c1a223   sw       t3,4(gp)
RVTEST_CODE_END
RVMODEL_HALT     # Writes successful completion to HTIF
# 80003220:          00408093   addi     ra,ra,4
# 80003224:          00100093   li       ra,1
# 80003228 <write_tohost>:
# 80003228:          00001f17   auipc    t5,0x1
# 8000322c:          dc1f2c23   sw       ra,-552(t5) # 80004000 <tohost>
# 80003230 <self_loop>:
# 80003230:          0000006f   j        80003230 <self_loop>
RVTEST_DATA_BEGIN    # Room for initialized data values (for loads and stores)
.align 4
rvtest_data:
.word 0xbabecafe
RVTEST_DATA_END
RVMODEL_DATA_BEGIN  # Start of signature
signature_x3_1:
    .fill 2*(XLEN/32),4,0xdeadbeef  # room for two 32-bit results
RVMODEL_DATA_END    # End of signature
```

Reference Output

rv32i_m/references/add-01.reference_output

```
80000000
00040000
```

How to Add a New Test

- **Recommended:** Use existing test as a template:
  - Replace the code between `RVTEST_SIGBASE` and `RVTEST_CODE_END`
  - Increase the amount of signature storage at the .fill statement as necessary
  - Change the `reference_output` file to match

- Add your new assembly language file to:

`wally-riscv-arch-test/riscv-test-suite/rv{32/64}i_m/<testtype>/src`

- Add your expected results and put it in the references directory
- Add test case name to the `Makefrag` list

Test Case Generator

Example test case generator:

`wally-riscv/tests/testgen/testgen.py`

- **It creates:**
  - `.S` test code
  - `.reference_output`
  - Expected values in:

`wally-riscv-arch-test/riscv-test-suite/rv{32,64}i_m/I`

- **It can be readily modified to generate more test cases.**
  - Add the names of the new test cases to the Makefrag list in the I directory and run make to rebuild with the new tests.

About these Notes

RISC-V System-on-Chip Design Lecture Notes