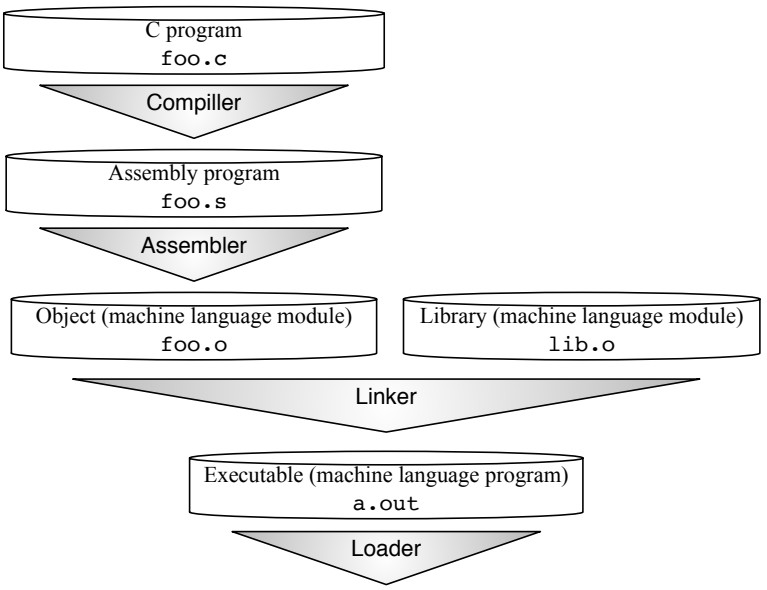

Figure 3.1: Steps of translation from C source code to a running program. These are the logical steps, although some steps are combined to accelerate translation. We use the Unix file suffix name convention for each type of file. The equivalent suffixes in MS-DOS are `.C`, `.ASM`, `.OBJ`, `.LIB`, and `.EXE`.

then the program must save register values in memory, but a surprising fraction of function calls fall into this happy case.

Other registers within a function call must be considered either in the same class as saved registers, which are preserved across a function call, or in the same class as the temporary registers, which are not. A function will change the register(s) containing the return value(s), so they are like temporary registers. There is no reason to preserve the registers to pass arguments to functions, so they also are like temporaries. The caller can rely on the remaining registers to be unchanged by across a function call: the registers used for the return address and the stack pointer. Figure 3.2 lists the RISC-V application binary interface (ABI) names of registers and the convention on whether they are preserved or not across function calls.

Given the ABI conventions, we can see the standard RV32I code for function entry and exit. The function *prologue* looks like this:

```
entry_label:
  addi sp,sp,-framesize    # Allocate space for stack frame
                           # by adjusting stack pointer (sp register)
  sw   ra,framesize-4(sp)  # Save return address (ra register)
  # save other registers to stack if needed
  ... # body of the function
```

If there are too many function arguments and variables to fit in the registers, the prologue allocates space on the stack for the function *frame*, as it is called. After the task of the function is complete, the *epilogue* undoes the stack frame and returns to the point of origin:

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No
f0–7	ft0–7	FP temporaries	No
f8–9	fs0–1	FP saved registers	Yes
f10–11	fa0–1	FP arguments/return values	No
f12–17	fa2–7	FP arguments	No
f18–27	fs2–11	FP saved registers	Yes
f28–31	ft8–11	FP temporaries	No

Figure 3.2: Assembler mnemonics for RISC-V integer and floating-point registers. RISC-V has enough registers that the ABI can allocate registers that procedures or methods are free to use without saving or restoring when they don't call other procedures or methods themselves. The registers preserved across a procedure call are also named *caller saved* versus *callee saved* for those that aren't. Chapter 5 explains the floating-point `f` registers. (Table 20.1 of [Waterman and Asanović 2017] is the basis of this figure.)

```
# restore registers from stack if needed
lw   ra,framesize-4(sp)  # Restore return address register
addi sp,sp, framesize    # De-allocate space for stack frame
ret                      # Return to calling point
```

We'll see an example that follows this ABI shortly, but first we need to explain the remaining assembly tasks beyond turning the ABI register names into register numbers.

■ ***Elaboration: The saved and temporary registers aren't contiguous***

to support RV32E, an embedded version of RISC-V that has only 16 registers (see Chapter 11). It simply uses register numbers `x0` to `x15`, so some saved and temporary registers are in this range, and the rest are in the last 16 registers. RV32E is smaller, but has no compiler support yet, since it doesn't match RV32I.

3.3 Assembly

The input to this step in Unix is a file with the suffix `.s`, such as `foo.s`; for MS-DOS it is `.ASM`.

The job of the assembler step of Figure 3.1 is not simply to produce object code from the instructions that the processor understands, but to extend them to include operations useful for the assembly language programmer or the compiler writer. This category, based on clever configurations of regular instructions, is called *pseudoinstructions*. Figures 3.3 and 3.4 list the RISC-V pseudoinstructions, with those in the first figure all relying on register x0 to always be zero while those in the second list do not. For example, `ret` mentioned above is actually a pseudoinstruction that the assembler replaces with `jalr x0, x1, 0` (see Figure 3.3). The majority of RISC-V pseudoinstructions depend on x0. As you can see, setting aside one of the 32 registers to be hardwired to zero greatly simplifies the RISC-V instruction set by providing many popular operations—such as jump, return, and branch on equal to zero—as pseudoinstructions.

Figure 3.5 shows the classic "Hello world" program in C. The compiler produces the assembly language output in Figure 3.6 using the calling convention in Figure 3.2 and the pseudoinstructions from Figures 3.3 and 3.4.

The "Hello world" program is typically the first program run on a newly designed processor. Architects traditionally consider running the operating system well enough to print "Hello world" as a strong sign that their new chip largely works. They email this output to their management and colleagues, and then they celebrate.

The commands that start with a period are *assembler directives*. They are commands to the assembler rather than code to be translated by it. They tell the assembler where to place code and data, specify text and data constants for use in the program, and so forth. Figure 3.9 shows the assembler directives of RISC-V. For Figure 3.6, the directives are:

- `.text`—Enter text section.
- `.align 2`—Align following code to 2^2 bytes.
- `.globl main`—Declare global symbol "main".
- `.section .rodata`—Enter read-only data section.
- `.balign 4`—Align data section to 4 bytes.
- `.string "Hello, %s!\n"`—Create this null-terminated string.
- `.string "world"`—Create this null-terminated string.

The assembler produces the object file in Figure 3.7 using the Executable and Linkable Format (ELF) standard format [TIS Committee 1995].

Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi x0, x0, 0`	No operation
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`j offset`	`jal x0, offset`	Jump
`jr rs`	`jalr x0, rs, 0`	Jump register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags

Figure 3.3: 32 RISC-V pseudoinstructions that rely on x0, the zero register. Appendix A includes includes the RISC-V pseudoinstructions as well as the real instructions. Those that read the 64-bit counters can read by upper 32 bits in RV32I by using the "h" version of the instructions and the lower 32 bits using the plain version. (Tables 20.2 and 20.3 of [Waterman and Asanović 2017] are the basis of this figure.).

Pseudoinstruction	Base Instruction(s)	Meaning
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address
`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
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`jal offset`	`jal x1, offset`	Jump and link
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags

Figure 3.4: 28 RISC-V pseudoinstructions that are independent of x0, the zero register. For `la`, GOT stands for Global Offset Table, which holds the runtime address of symbols in dynamically linked libraries. Appendix A includes the RISC-V pseudoinstructions as well as the real instructions. (Tables 20.2 and 20.3 of [Waterman and Asanović 2017] are the basis of this figure.)

```
#include <stdio.h>
int main()
{
   printf("Hello, %s\n", "world");
   return 0;
}
```

Figure 3.5: Hello World program in C (`hello.c`).

```
  .text                    # Directive: enter text section
  .align 2                 # Directive: align code to 2^2 bytes
  .globl main              # Directive: declare global symbol main
main:                      # label for start of main
  addi sp,sp,-16           # allocate stack frame
  sw   ra,12(sp)           # save return address
  lui  a0,%hi(string1)     # compute address of
  addi a0,a0,%lo(string1)  #   string1
  lui  a1,%hi(string2)     # compute address of
  addi a1,a1,%lo(string2)  #   string2
  call printf              # call function printf
  lw   ra,12(sp)           # restore return address
  addi sp,sp,16            # deallocate stack frame
  li   a0,0                # load return value 0
  ret                      # return
  .section .rodata         # Directive: enter read-only data section
  .balign 4                # Directive: align data section to 4 bytes
string1:                   # label for first string
  .string "Hello, %s!\n"   # Directive: null-terminated string
string2:                   # label for second string
  .string "world"          # Directive: null-terminated string
```

Figure 3.6: Hello World program in RISC-V assembly language (`hello.s`).

```
00000000 <main>:
 0: ff010113  addi  sp,sp,-16
 4: 00112623  sw    ra,12(sp)
 8: 00000537  lui   a0,0x0
 c: 00050513  mv    a0,a0
10: 000005b7  lui   a1,0x0
14: 00058593  mv    a1,a1
18: 00000097  auipc ra,0x0
1c: 000080e7  jalr  ra
20: 00c12083  lw    ra,12(sp)
24: 01010113  addi  sp,sp,16
28: 00000513  li    a0,0
2c: 00008067  ret
```

Figure 3.7: Hello World program in RISC-V machine language (`hello.o`). The six instructions that are later patched by the linker (locations 8 to 1c) have zero in their address fields. The symbol table included in the object file records the labels and addresses of all the instructions that need to be edited by the linker.

```
000101b0 <main>:
   101b0: ff010113 addi sp,sp,-16
   101b4: 00112623 sw   ra,12(sp)
   101b8: 00021537 lui  a0,0x21
   101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
   101c0: 000215b7 lui  a1,0x21
   101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
   101c8: 288000ef jal  ra,10450 <printf>
   101cc: 00c12083 lw   ra,12(sp)
   101d0: 01010113 addi sp,sp,16
   101d4: 00000513 li   a0,0
   101d8: 00008067 ret
```

Figure 3.8: Hello World program as RISC-V machine language program after linking. In Unix systems, the file would be named `a.out`.

Directive	Description
`.text`	Subsequent items are stored in the text section (machine code).
`.data`	Subsequent items are stored in the data section (global variables).
`.bss`	Subsequent items are stored in the bss section (global variables initialized to 0).
`.section .foo`	Subsequent items are stored in the section named `.foo`.
`.align n`	Align the next datum on a 2^n-byte boundary. For example, `.align 2` aligns the next value on a word boundary.
`.balign n`	Align the next datum on a n-byte boundary. For example, `.balign 4` aligns the next value on a word boundary.
`.globl sym`	Declare that label sym is global and may be referenced from other files.
`.string "str"`	Store the string str in memory and null-terminate it.
`.byte b1,..., bn`	Store the n 8-bit quantities in successive bytes of memory.
`.half w1,...,wn`	Store the n 16-bit quantities in successive memory halfwords.
`.word w1,...,wn`	Store the n 32-bit quantities in successive memory words.
`.dword w1,...,wn`	Store the n 64-bit quantities in successive memory doublewords.
`.float f1,..., fn`	Store the n single-precision floating-point numbers in successive memory words.
`.double d1,..., dn`	Store the n double-precision floating-point numbers in successive memory doublewords.
`.option rvc`	Compress subsequent instructions (see Chapter 7).
`.option norvc`	Don't compress subsequent instructions.
`.option relax`	Allow linker relaxations for subsequent instructions.
`.option norelax`	Don't allow linker relaxations for subsequent instructions.
`.option pic`	Subsequent instructions are position-independent code.
`.option nopic`	Subsequent instructions are position-dependent code.
`.option push`	Push the current setting of all `.option`s to a stack, so that a subsequent `.option pop` will restore their value.
`.option pop`	Pop the option stack, restoring all `.option`s to their setting at the time of the last `.option push`.

Figure 3.9: Common RISC-V assembler directives.

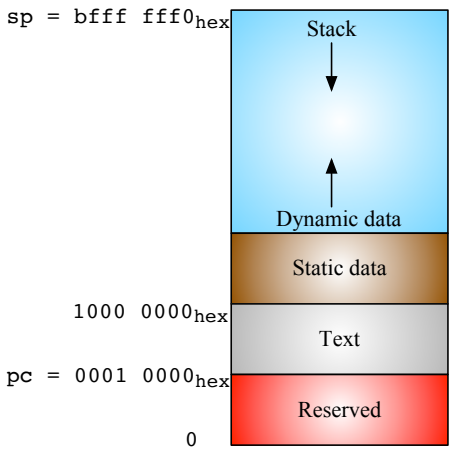

Figure 3.10: RV32I allocation of memory to program and data. The high addresses are the top of the figure and the low addresses are the bottom. In this RISC-V software convention, the stack pointer (`sp`) starts at bfff fff0$_{hex}$ and grows down toward the Static data. The *text* (program code) starts at 0001 0000$_{hex}$ and includes the statically-linked libraries. The Static data starts immediately above the text region; in this example, we assume that address is 1000 0000$_{hex}$. Dynamic data, allocated in C by `malloc()`, is just above the Static data. Called the `heap`, it grows upward toward the stack. It includes the dynamically-linked libraries.

3.4 Linker

Rather than compile all the source code every time one file changes, the linker allows individual files to be compiled and assembled separately. It "stitches" the new object code together with existing machine language modules, such as libraries. It derives its name from one of its tasks, which is to all edit the links of the jump and link instructions in the object file. In fact, linker is short for link editor, which was the historical name of this step in Figure 3.1. In Unix systems, the input to the linker are files with the `.o` suffix (e.g., `foo.o, libc.o`), and its output is the `a.out` file. For MS-DOS, the inputs are files with the suffix `.OBJ` or `.LIB` and the output is a `.EXE` file.

Figure 3.10 shows the addresses of the regions of memory allocated for code and data in a typical RISC-V program. The linker must adjust the program and data addresses of instructions in all the object files to match addresses in this figure. It is less work for the linker if the input files are *position independent code* (*PIC*). PIC means that all the branches to instructions and references to data inside the file are correct wherever the code is placed. As mentioned in Chapter 2, the PC-relative branch of RV32I makes PIC much easier to fulfill.

In addition to the instructions, each object file contains a symbol table that includes all the labels in the program that must be given addresses as part of the linking process. This list includes labels to data as well as to code. Figure 3.6 has two data labels to be set (`string1` and `string2`) and two code labels to be assigned in (`main` and `printf`). Since it's hard to specify a 32-bit address within a single 32-bit instruction, the linker must adjust two instructions per label in the RV32I code, as Figure 3.6 shows: `lui` and `addi` for data addresses, and

`auipc` and `jalr` for code addresses. Figure 3.8 shows the final linked `a.out` version of the object file in Figure 3.7.

RISC-V compilers support several ABIs, depending on whether the F and D extensions are present. For RV32, the ABIs are named ilp32, ilp32f, and ilp32d. ilp32 means that the C language data types int, long, and pointer are all 32 bits; the optional suffix indicates how floating-point arguments are passed. In ilp32, floating-point arguments are passed in integer registers. In ilp32f, single-precision floating-point arguments are passed in floating-point registers. In ilp32d, double-precision floating-point arguments are also passed in floating-point registers.

Naturally, to pass a floating-point argument in a floating-point register, you need the corresponding floating-point ISA extension F or D (see Chapter 5). So, to compile code for RV32I (GCC flag '-march=rv32i'), you must use the ilp32 ABI (GCC flag '-mabi=ilp32'). On the other hand, having floating-point instructions doesn't mean the calling convention is required to use them; so, for example, RV32IFD is compatible with all three ABIs: ilp32, ilp32f, and ilp32d.

The linker checks that the program's ABI matches all of its libraries. Although the compiler supports many combinations of ISA extensions and ABIs, only a few sets of libraries might be installed. Hence, a common pitfall is attempting to link a program without having compatible libraries installed. The linker will not produce a helpful diagnostic message in this case; it will simply attempt to link with an incompatible library, then inform you of the incompatibility. This pitfall generally occurs only when compiling on one computer for a different computer (*cross compiling*).

■ ***Elaboration: Linker relaxation***

The jump and link instruction has a 20-bit PC-relative address field, so a single instruction can jump far. While the compiler produces two instructions for each external function, quite often only one instruction is necessary. Since this optimization saves both time and space, linkers will make passes over the code to replace two instructions with one whenever it can. Because a pass might shrink distance between a call and the function so that it now fits in a single instruction, the linker keeps optimizing the code until there are no further changes. This process is called *Linker relaxation*, with the name referring to relaxation techniques for solving systems of equations. In addition to procedure calls, the RISC-V linker relaxes data addressing to use the global pointer when the datum lies within ± 2 KiB of `gp`, removing a `lui` or `auipc`. It similarly relaxes thread-local storage addressing when the datum lies within ± 2 KiB of `tp`.

3.5 Static vs. Dynamic Linking

Architects typically measure processor performance using benchmarks that are statically linked despite most real programs having dynamic links. The excuse is that users interested in performance should link statically, but it's a poor justification. It makes more sense to accelerate performance of real programs, not benchmarks.

The prior section describes *static linking*, where all potential library code is linked and then loaded together before execution. Such libraries can be relatively large, so linking a popular library into multiple programs wastes memory. Moreover, the libraries are bound when linked—even when they are updated later to fix bugs—forcing the statically-linked code to use the old, buggy version.

To avoid both problems, most systems today rely on *dynamic linking*, where the desired external function is loaded and linked to the program only after it is first called; if it's never called, it's never loaded and linked. Every call after the first one uses a fast link, so the dynamic overhead is only paid once. Each time a program starts it links in the current version

of the library functions it needs, which is how it can get the newest version. Furthermore, if multiple programs use the same dynamically linked library, the code in the library need appear only once in memory.

The code that the compiler generates resembles that for static linking. Instead of jumping to the real function, it jumps to a short (three-instruction) stub function. The stub function loads the address of the real function from a table in memory, then jumps to it. However, on the first call, the table lacks the address of the real function, but instead contains the address of the dynamic-linking routine. When invoked, the dynamic linker uses the symbol table to find the real function, copies it into memory, and then updates the table to point to the real function. Each subsequent call pays only the three-instruction overhead of the stub function.

3.6 Loader

A program like the one in Figure 3.8 is an executable file kept in the computer's storage. When one is to be run, the loader's job is to load it into memory and jump to the starting address. The "loader" today is the operating system; stated alternatively, loading `a.out` is one of many tasks of an operating system.

Loading is a little trickier for dynamically-linked programs. Instead of simply starting the program, the operating system starts the dynamic linker. It in turn starts the desired program, and then handles all first-time external calls, copies the functions into memory, and edits the program after each call to point it to the correct function.

3.7 Concluding Remarks

> *Keep it simple, stupid.*
>
> —Kelly Johnson, aeronautical engineer who coined the "KISS Principle," 1960

The assembler enhances the simple RISC-V ISA with 60 pseudoinstructions that make RISC-V code easier to read and to write without increasing hardware costs. Simply dedicating one RISC-V register to zero enables many of these helpful operations. The Load Upper Immediate (`lui`) and Add Upper Immediate to PC (`auipc`) instructions make it easier for the compiler and linker to adjust addresses for external data and functions, and PC-relative branching makes it easier to help the linker with position-independent code. Having plenty of registers enables a calling convention that makes function call and return faster by reducing the number of register spills and restores.

RISC-V offers a tasteful collection of simple but impactful mechanisms that reduce cost, improve performance, and make it easier to program.

3.8 To Learn More

D. A. Patterson and J. L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

TIS Committee. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2. *TIS Committee*, 1995.

A. Waterman and K. Asanović, editors. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*. May 2017. URL `https://riscv.org/specifications/`.

Notes

[1] `http://parlab.eecs.berkeley.edu`

4 RV32M: Multiply and Divide

Entities should not be multiplied beyond necessity.

—William of Occam, 1320

William of Occam (1287-1347) was an English theologian who promoted what is now called "Occam's razor," a preference for simplicity in the scientific method.

4.1 Introduction

RV32M adds integer multiply and divide instructions to RV32I. Figure 4.1 is a graphical representation of the RV32M extension instruction set and Figure 4.2 lists their opcodes.

Divide is straightforward. Recall that

$$Quotient = (Dividend - Remainder) \div Divisor$$

or alternatively

$$Dividend = Quotient \times Divisor + Remainder$$

$$Remainder = Dividend - (Quotient \times Divisor)$$

`srl` can do unsigned division by 2^i. For example, if a2 = 16 (2^4) then `srli t2,a1,4` produces the same value as `divu t2,a1,a2`.

RV32M has divide instructions for both signed and unsigned integers: divide (`div`) and divide unsigned (`divu`), which place the quotient into the destination register. Less frequently, programmers want the remainder instead of the quotient, so RV32M offers remainder (`rem`) and remainder unsigned (`remu`), which write the remainder instead of the quotient.

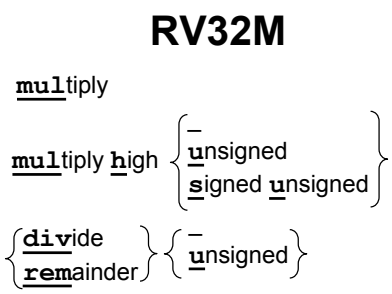

Figure 4.1: Diagram of the RV32M instructions.