CS61C : Machine Structures

Lecture 3.2.2

Floating Point II & Linking

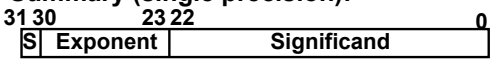
2004-07-06

Kurt Meinz

inst.eecs.berkeley.edu/~cs61c

FP Review

- Floating Point numbers approximate values that we want to use.
- IEEE 754 Floating Point Standard is most widely accepted attempt to standardize interpretation of such numbers
  - Every desktop or server computer sold since ~1997 follows these conventions
- Summary (single precision):



- $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
- Double precision identical, bias of 1023

Representation for Denorms (1/3)

- Problem: There's a gap among representable FP numbers around 0
  - Smallest representable pos num:
    - $a = 1.0\ldots_2 * 2^{-126} = 2^{-126}$
  - Second smallest representable pos num:
    - $b = 1.000\ldots\ldots1_2 * 2^{-126} = 2^{-126} + 2^{-149}$

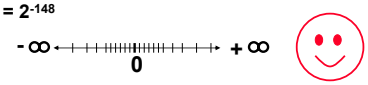
$a - 0 = 2^{-126}$

$b - a = 2^{-149}$

Normalization and implicit 1 is to blame!

Representation for Denorms (2/3)

- Solution:
  - We still haven't used Exponent = 0, Significand nonzero
  - Denormalized number: no leading 1, implicit exponent = -126.
  - Smallest representable pos num:
    - $a = 2^{-149}$
  - Second smallest representable pos num:
    - $b = 2^{-148}$



Representation for Denorms (3/3)

- Normal FP equation:
  - $(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$
- If (fp.exp == 0 and fp.signifcant != 0)
  - Denorm
  - $(-1)^S \times (0 + \text{Significand}) \times 2^{(-126)}$

IEEE Four Rounding Modes

- Math on real numbers ⇒ we worry about rounding to fit result in the significant field.
- FP hardware carries 2 extra bits of precision, and rounds for proper value
- Rounding occurs when converting…
  - double to single precision
  - floating point # to an integer

IEEE Four Rounding Modes

- Round towards + ∞
  - ALWAYS round "up": 2.1 ⇒ 3, -2.1 ⇒ -2
- Round towards - ∞
  - ALWAYS round "down": 1.9 ⇒ 1, -1.9 ⇒ -2
- Truncate
  - Just drop the last bits (round towards 0)
- Round to (nearest) even (default)
  - Normal rounding, almost: 2.5 ⇒ 2, 3.5 ⇒ 4
  - Like you learned in grade school
  - Insures fairness on calculation
  - Half the time we round up, other half down

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
Multiplicand   1000        8
 Multiplier   x1001        9
               1000
              0000
             0000
            +1000
            01001000
```

- m bits x n bits = m + n bit product

Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - `mult` register1, register2
  - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
    - puts product upper half in hi, lower half in lo
  - hi and lo are 2 registers separate from the 32 general purpose registers
  - Use `mfhi` register & `mflo` register to move from hi, lo to another register

Integer Multiplication (3/3)

- Example:
  - in C: `a = b * c;`
  - in MIPS:
    - let b be $s2; let c be $s3; and let a be $s0 and $s1 (since it may be up to 64 bits)

```
mult $s2,$s3   # b*c
mfhi $s0       # upper half of
               # product into $s0
mflo $s1       # lower half of
               # product into $s1
```

- Note: Often, we only care about the lower half of the product.

Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                 1001   Quotient
Divisor 1000|1001010   Dividend
           -1000
               10
               101
               1010
              -1000
                 10   Remainder
                    (or Modulo result)
```

- Dividend = Quotient x Divisor + Remainder

Integer Division (2/2)

- Syntax of Division (signed):
  - `div` register1, register2
  - Divides 32-bit register 1 by 32-bit register 2:
  - puts remainder of division in `hi`, quotient in `lo`
- Implements C division (`/`) and modulo (%)
- Example in C:

```
a = c / d;
b = c % d;
```

- in MIPS: a↔$s0;b↔$s1;c↔$s2;d↔$s3

```
div  $s2,$s3    # lo=c/d, hi=c%d
mflo $s0        # get quotient
mfhi $s1        # get remainder
```

Unsigned Instructions & Overflow

- MIPS also has versions of `mult`, `div` for unsigned operands:
  - `multu`
  - `divu`
  - Determines whether or not the product and quotient are changed if the operands are signed or unsigned.
- MIPS does not check overflow on ANY signed/unsigned multiply, divide instr
  - Up to the software to check `hi`

FP Addition & Subtraction

- Much more difficult than with integers (can't just add significands)
- How do we do it?
  - De-normalize to match larger exponent
  - Add significands to get resulting one
  - Normalize (& check for under/overflow)
  - Round if needed (may need to renormalize)
- If signs ≠, do a subtract. (Subtract similar)
  - If signs ≠ for add (or = for sub), what's ans sign?
- Question: How do we integrate this into the integer arithmetic unit? [Answer: We don't!]

MIPS Floating Point Architecture (1/4)

- Separate floating point instructions:
  - Single Precision:
    `add.s, sub.s, mul.s, div.s`
  - Double Precision:
    `add.d, sub.d, mul.d, div.d`
- These are far more complicated than their integer counterparts
  - Can take much longer to execute

MIPS Floating Point Architecture (2/4)

- Problems:
  - Inefficient to have different instructions take vastly differing amounts of time.
  - Generally, a particular piece of data will not change FP ⇔ `int` within a program.
    - Only 1 type of instruction will be used on it.
  - Some programs do no FP calculations
  - It takes lots of hardware relative to integers to do FP fast

MIPS Floating Point Architecture (3/4)

- 1990 Solution: Make a completely separate chip that handles only FP.
- Coprocessor 1: FP chip
  - contains 32 32-bit registers: `$f0`, `$f1`, …
  - most of the registers specified in `.s` and `.d` instruction refer to this set
  - separate load and store: `lwc1` and `swc1` ("load word coprocessor 1", "store …")
  - Double Precision: by convention, even/odd pair contain one DP FP number: `$f0`/`$f1`, `$f2`/`$f3`, … , `$f30`/`$f31`
    - Even register is the name

MIPS Floating Point Architecture (4/4)

- 1990 Computer actually contains multiple separate chips:
  - Processor: handles all the normal stuff
  - Coprocessor 1: handles FP and only FP;
  - more coprocessors?… Yes, later
  - Today, FP coprocessor integrated with CPU, or cheap chips may leave out FP HW
- Instructions to move data between main processor and coprocessors:
  - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.
- Appendix pages A-70 to A-74 contain many, many more FP operations.

Questions

1. Converting `float` -> `int` -> `float` produces same `float` number
   No! 3.14 -> 3 -> 3
2. Converting `int` -> `float` -> `int` produces same `int` number
   No! (2^30 + 2^1)
3. FP add is associative: `(x+y)+z = x+(y+z)`
   No! 2^30 + -2^30 + 1

FP/Math Summary

- Reserve exponents, significands:

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN

- Integer `mult`, `div` uses `hi`, `lo` regs
  - `mfhi` and `mflo` copies out.
- Four rounding modes (to even default)
- MIPS FL ops complicated, expensive

CLL Overview

- Interpretation vs Translation
- Translating C Programs
  - Compiler
  - Assembler
  - Linker
  - Loader
- An Example

Language Continuum

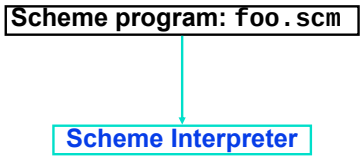
Scheme
Java C++ C Assembly machine language

Easy to write
Inefficient to run

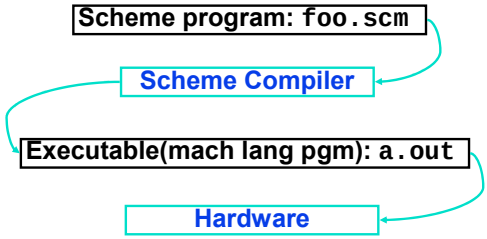
Difficult to write
Efficient to run

- Interpret a high level language if efficiency is not critical
- Translate (compile) to a lower level language to improve performance
- Scheme example …

Interpretation



Translation



° Scheme Compiler is a translator from Scheme to machine language.

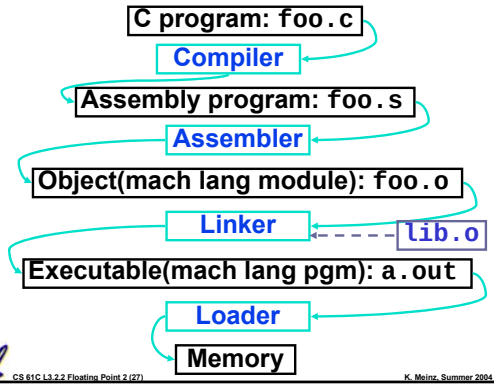
Interpretation

- Any good reason to interpret machine language in software?
- SPIM – useful for learning / debugging
- Apple Macintosh conversion
  - Switched from Motorola 680x0 instruction architecture to PowerPC.
  - Could require all programs to be re-translated from high level language
  - Instead, let executables contain old and/or new machine code, interpret old code in software if necessary

Interpretation vs. Translation?

- Easier to write interpreter
- Interpreter closer to high-level, so gives better error messages (e.g., SPIM)
  - Translator reaction: add extra information to help debugging (line numbers, names)
- Interpreter slower (10x?) but code is smaller (1.5X to 2X?)

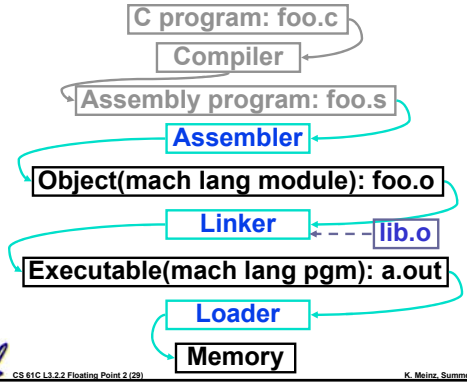
Steps to Starting a Program



Compiler

- Input: High-Level Language Code (e.g., C, Java such as `foo.c`)
- Output: MAL Assembly Language Code (e.g., `foo.s` for MIPS)
- Note: Output *may* contain pseudoinstructions
- *(btw: hardest stage by far!)*

Where Are We Now?



Assembler

- Input: MAL Assembly Language Code (e.g., `foo.s` for MIPS)
- Output: Object Code, information tables (e.g., `foo.o` for MIPS)
- Reads and Uses Directives
- Replace Pseudoinstructions
- Produce Machine Language
- Creates Object File

Assembler Directives (p. A-51 to A-53)

- Give directions to assembler, but do not produce machine instructions
  - `.text`: Subsequent items put in user text segment
  - `.data`: Subsequent items put in user data segment
  - `.globl sym`: declares `sym` global and can be referenced from other files
  - `.asciiz str`: Store the string `str` in memory and null-terminate it
  - `.word w1…wn`: Store the *n* 32-bit quantities in successive memory words

Pseudoinstruction Replacement

- Asm. treats convenient variations of machine language instructions as if real instructions

Pseudo:	Real:
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	`sw $a0, 32($sp)` `sw $a1, 36($sp)`
`mul $t7,$t6,$t5`	`mult $t6,$t5` `mflo $t7`
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`ble $t0,100,loop`	`slti $at,$t0,101` `bne $at,$0,loop`
`la $a0, str`	`lui $at,left(str)` `ori $a0,$at,right(str)`

Producing Machine Language (1/3)

- Constraint on Assembler:
  - The object file output (foo.o) may be only one of many object files in the final executable:
    - C: #include “my_helpers.h”
    - C: #include <stdio.h>
- Consequences:
  - Object files won’t know their base addresses until they are linked/loaded!
  - References to addresses will have to be adjusted in later stages

Producing Machine Language (2/3)

- Simple Case
  - Arithmetic, Logical, Shifts, and so on.
  - All necessary info is within the instruction already.
- What about Branches?
  - PC-Relative and in-file
  - In TAL, we know by how many instructions to branch.
- So these can be handled easily.

Producing Machine Language (3/3)

- What about jumps (`j` and `jal`)?
  - Jumps require absolute address.
- What about references to data?
  - `la` gets broken up into `lui` and `ori`
  - These will require the full 32-bit address of the data.
- These can’t be determined yet, so we create two tables for use by linker/loader…

1: Symbol Table

- List of “items” provided by this file.
  - What are they?
    - Labels: function calling
    - Data: anything in the `.data` section; variables which may be accessed across files
  - Includes base address of label in the file.

2: Relocation Table

- List of "items" needed by this file.
  - Any label jumped to: `j` or `jal`
    - internal
    - external (including lib files)
  - Any named piece of data
    - Anything referenced by the `la` instruction
    - static variables
  - Contains base address of instruction w/dependency, dependency name

Question

- Which lines go in the symbol table and/or relocation table?

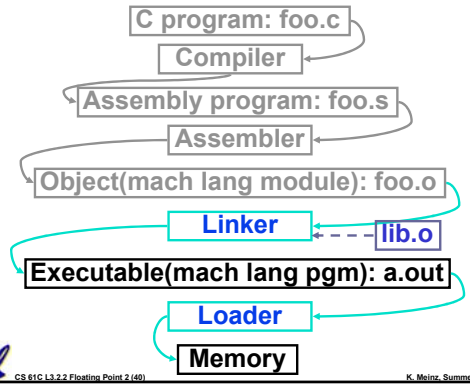
```
my_func:
   lui $a0 my_arrayh       # a (from la)
   ori $a0 $a0 my_arrayl   # b (from la)
   jal add_link            # c
   bne $a0,$v0, my_func    # d
```

A:	Symbol: my_func	relocate: my_array
B:	-	relocate: my_array
C:	-	relocate: add_link
D:	-	-

Object File Format

- object file header: size and position of the other pieces of the object file
- text segment: the machine code
- data segment: binary representation of the data in the source file
- relocation information: identifies lines of code that need to be "handled"
- symbol table: list of this file's labels and data that can be referenced
- debugging information

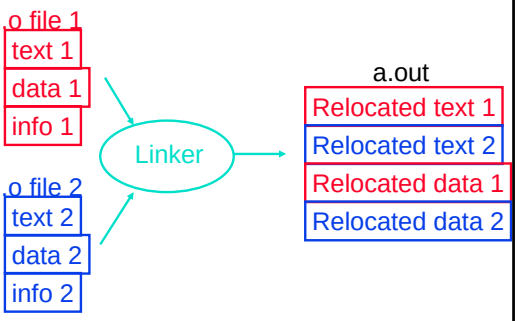
Where Are We Now?



Link Editor/Linker (1/3)

- Input: Object Code, information tables (e.g., `foo.o` for MIPS)
- Output: Executable Code (e.g., `a.out` for MIPS)
- Combines several object (.o) files into a single executable ("linking")
- Enable Separate Compilation of files
  - Changes to one file do not require recompilation of whole program
    - Windows NT source is >40 M lines of code!
  - Link Editor name from editing the "links" in jump and link instructions

Link Editor/Linker (2/3)



Link Editor/Linker (3/3)

- Step 1: Take text segment from each .o file and put them together.
- Step 2: Take data segment from each .o file, put them together, and concatenate this onto end of text segments.
- Step 3: Resolve References
  - Go through Relocation Table and handle each entry
  - That is, fill in all absolute addresses

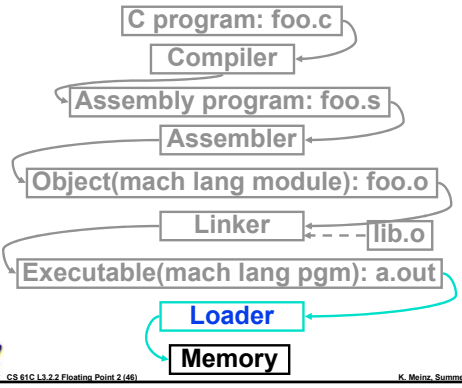
Resolving References (1/2)

- Linker *assumes* first word of first text segment is at address 0x00000000.
- Linker knows:
  - length of each text and data segment
  - ordering of text and data segments
- Linker calculates:
  - absolute address of each label to be jumped to (internal or external) and each piece of data being referenced

Resolving References (2/2)

- To resolve references:
  - search for reference (data or label) in all symbol tables
  - if not found, search library files (for example, for `printf`)
  - once absolute address is determined, fill in the machine code appropriately
- Output of linker: executable file containing text and data (plus header)

Where Are We Now?



Loader (1/3)

- Input: Executable Code (e.g., `a.out` for MIPS)
- Output: (program is run)
- Executable files are stored on disk.
- When one is run, loader's job is to load it into memory and start it running.
- In reality, loader is the operating system (OS)
  - loading is one of the OS tasks

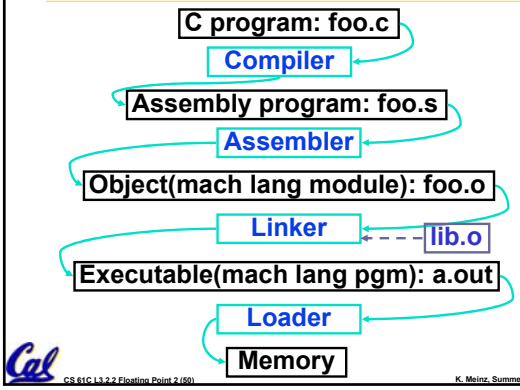
Loader (2/3)

- So what does a loader do?
- Reads executable file's header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space (this may be anywhere in memory)

Loader (3/3)

- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
- Jumps to start-up routine that copies program's arguments from stack to registers and sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call

Things to Remember (1/3)



Things to Remember (2/3)

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudoinstructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). This changes each .s file into a .o file.
- Linker combines several .o files and resolves absolute addresses.
- Loader loads executable into memory and begins execution.

Things to Remember 3/3

- Stored Program concept mean instructions just like data, so can take data from storage, and keep transforming it until load registers and jump to routine to begin execution
  - Compiler ⇒ Assembler ⇒ Linker (⇒ Loader)
- Assembler does 2 passes to resolve addresses, handling internal forward references
- Linker enables separate compilation, libraries that need not be compiled, and resolves remaining addresses

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
#include <stdio.h>

int main (int argc, char *argv[]) {

 int i;

 int sum = 0;


 for (i = 0; i <= 100; i = i + 1)
    sum = sum + i * i;

 printf ("The sum from 0 .. 100 is %d\n",
    sum);

}
```

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
_.text
 .align  2
 .globl  main
main:
 subu $sp,$sp,32
 sw  $ra, 20($sp)
 sd  $a0, 32($sp)
 sw  $0, 24($sp)
 sw  $0, 28($sp)
loop:
 lw  $t6, 28($sp)
 mul $t7, $t6,$t6
 lw  $t8, 24($sp)
 addu $t9,$t8,$t7
 sw  $t9, 24($sp)
 addu $t0, $t6, 1
 sw  $t0, 28($sp)
 ble $t0,100, loop
 la  $a0, str
 lw  $a1, 24($sp)
 jal printf
 move $v0, $0
 lw  $ra, 20($sp)
 addiu $sp,$sp,32
 j   $ra
 .data
 .align  0
str:
 .asciiz "The sum from 0 .. 100 is %d\n"
```

Where are 7 pseudo-instructions?

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
_.text
 .align  2
 .globl  main
main:
 subu $sp,$sp,32
 sw  $ra, 20($sp)
 sd  $a0, 32($sp)
 sw  $0, 24($sp)
 sw  $0, 28($sp)
loop:
 lw  $t6, 28($sp)
 mul $t7, $t6,$t6
 lw  $t8, 24($sp)
 addu $t9,$t8,$t7
 sw  $t9, 24($sp)
```

```
 addu $t0, $t6, 1
 sw  $t0, 28($sp)
 ble $t0,100, loop
 la  $a0, str
 lw  $a1, 24($sp)
 jal printf
 move $v0, $0
 lw  $ra, 20($sp)
 addiu $sp,$sp,32
 j   $ra
 .data
 .align  0
str:
 .asciiz "The sum
 from 0 .. 100 is
 %d\n"
```

7 pseudo-instructions underlined

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

- Remove pseudoinstructions, assign addresses

```
00 addiu $29,$29,-32
04 sw    $31,20($29)
08 sw    $4, 32($29)
0c sw    $5, 36($29)
10 sw    $0, 24($29)
14 sw    $0, 28($29)
18 lw    $14, 28($29)
1c multu $14, $14
20 mflo  $15
24 lw    $24, 24($29)
28 addu  $25,$24,$15
2c sw    $25, 24($29)
```

```
30 addiu $8,$14, 1
34 sw    $8,28($29)
38 slti  $1,$8, 101
3c bne   $1,$0, -10
40 lui   $4, l.str
44 ori   $4,$4,r.str
48 lw    $5,24($29)
4c jal   printf
50 add   $2, $0, $0
54 lw    $31,20($29)
58 addiu $29,$29,32
5c jr    $31
```


Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

- Example.o contains these tables:
- Symbol Table
  - Label Address

```
main:   text+0x00000000 global
loop:   text+0x00000018
str:    data+0x00000000
```

- Relocation Information

Address	Instr. Type	Dependency
text+00040	lui	l.str
text+00044	ori	r.str
text+0004c	jal	printf

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

- Linker sees all the .o files.
  - One of these (example.o) provides main and needs printf.
  - Another (stdio.o) provides printf.
- 1) Linker decides order of text, data segments
- 2) This fills out the symbol tables
- 3) This fills out the relocation tables

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

- Linker first stage:
  - Set text= 0x0400 0000; data=0x1000 0000
- Symbol Table
  - Label Address

```
main:   0x04000000 global
loop:   0x04000018
str:    0x10000000
```

- Relocation Information

Address	Instr. Type	Dependency
text+0x0040	lui	l.str
text+0x0044	ori	r.str
text+0x004c	jal	printf

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

- Linker second stage:
  - Set text= 0x0400 0000; data=0x1000 0000
- Symbol Table
  - Label Address

```
main:   0x04000000 global
loop:   0x04000018
str:    0x10000000
```

- Relocation Information

Address	Instr. Type	Dependency
text+0x0040	lui	l.str=0x1000
text+0x0044	ori	r.str=0x0000
text+0x004c	jal	printf=04440000

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

- Edit Addresses: start at 0x0400000

```
00 addiu $29,$29,-32      30 addiu $8,$14, 1
04 sw    $31,20($29)      34 sw    $8,28($29)
08 sw    $4, 32($29)      38 slti  $1,$8, 101
0c sw    $5, 36($29)      3c bne   $1,$0, -10
10 sw    $0, 24($29)      40 lui   $4, 1000
14 sw    $0, 28($29)      44 ori   $4,$4,0000
18 lw   $14, 28($29)      48 lw    $5,24($29)
1c multu $14, $14         4c jal   01110000
20 mflo  $15              50 add   $2, $0, $0
24 lw   $24, 24($29)      54 lw    $31,20($29)
28 addu $25,$24,$15       58 addiu $29,$29,32
2c sw   $25, 24($29)      5c jr    $31
```

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

```
0x004000  00100111101111101111111111100000
0x004004  10101111101111110000000000010100
0x004008  10101111101001000000000000100000
0x00400c  10101111101001010000000000100100
0x004010  10101111101000000000000000011000
0x004014  10101111101000000000000000011100
0x004018  10001111101011100000000000011100
0x00401c  10001111101111000000000000011000
0x004020  00000001110011100000000000011001
0x004024  00100101110010000000000000000001
0x004028  00101001000000010000000001100101
0x00402c  10101111101010000000000000011100
0x004030  00000000000000000111100000010010
0x004034  00000011000011111100100000100001
0x004038  00010100001000001111111111110111
0x00403c  10101111101110010000000000011000
0x004040  00111100000001000001000000000000
0x004044  10001111101001010000000000011000
0x004048  00001100000010000000000000011101100
0x00404c  00100100100001000000010000110000
0x004050  10001111101111110000000000010100
0x004054  00100111101111101000000000100000
0x004058  00000011111000000000000000001000
0x00405c  00000000000000000000010000100001
```