The Haverford Educational RISC Architecture

Guide to Version 2.4

David G. Wonnacott

Table of contents

List of figures

Prologue

The Haverford Educational RISC Architecture (HERA) provides a foundation for the multi-course project that unifies Haverford's upper-level computer science curriculum. The HERA instruction set is powerful enough to illustrate assembly-language programming techniques and serve as a target for compilers, yet simple enough to be implemented as a student project and extended in hardware/software co-design projects to provide operating system support. Thus, across the lab projects of four classes, students produce a system in which machine-languge code they write or generate with a compiler can be executed on a microprocessor they have designed, and on which I/O to an ASCII terminal can be performed via device drivers they have written themselves. The Haverford curriculum couples these labs with lectures and smaller assignments that contrast HERA with real-world systems. See [Won06] for more details of the educational uses of HERA.

The HERA-C development system lets students execute HERA assembly language programs before their own system is operational. HERA-C is a set of C/C++ macros that lets students compile, execute, and debug HERA programs with a standard C++ development environment, minimizing distractions from new tools. See `www.cs.haverford.edu/software/HERA` for more information about HERA-C, the HERA assembler Hassem, and other supporting tools.

The HERA system originated with an attempt to adapt Andrew Appel's "Jouette" [App98] for use in a Computer Hardware course based on [Man88]. The current system owes much to the helpful criticism and patience of students who endured early versions. Todd Miller (Haverford College class of 2001) also contributed significantly to the early macros that became HERA-C. My thanks to all of you!

Organization of this document:

Part I focuses primarily on the hardware features provided by a HERA CPU and a standard assembler, without going into detail on the usage or reason for each feature. It is intended to serve as a primary reference for anyone building or extending the hardware of a HERA CPU, or for experienced assembly-language programmers wishing to pick up HERA.

Part II discusses typical usages of HERA hardware instructions and assembler features. It is intended to serve as a primary reference for those learning assembly-language programming for the first time, or for experienced assembly-language programmers who need a reference for the conventions of HERA (using the notation of the HERA-C system).

Part I

Definitions

Chapter 1
Architecture Overview

1.1 User-Addressable Registers

The HERA processor has sixteen 16-bit user-addressable registers, $R_0 ... R_{15}$, which can be used as operands or targets for most instructions (e.g., `ADD` or `LOAD`). Programs primarily use $R_1 ... R_{10}$ to hold information they are processing, much as high-level language programs use variables. In contrast, R_0 and R_{11}-R_{15} have specific definitions or conventional uses: The zero register (R_0) always has the value 0 (i.e., the processor ignores any value directed into it); the stack pointer (R_{15}, usually referred to as SP) and frame pointer (R_{14}, usually referred to as FP) are used to maintain stack frames that (along with $R_1 .... R_{10}$) hold the values of local variables; the other registers above R_{10} are, by convention, used by programmers and assemblers for specific purposes such as function calls, branches, and in other short-term roles, as detailed in Chapters 3 and 7.

1.2 Program Counter and Flags Register

The processor also has a 16-bit program counter (PC) and a set of status/control flags (sometimes referred to as *condition bits*). The flags are: sign (s, or F_0), which is set to true when a negative value is produced by most operations; zero (z, or F_1), which is set to true when zero is produced; overflow (v, or F_2), which indicates overflow from a *signed* arithmetic operation; and carry (c, or F_3), which indicates overflow from an *unsigned* operation.

These flags control the behavior of conditional branch instructions; the value of the carry flag may also be used in subsequent arithmetic operations, to support multiple-precision calculations. There is an additional 5th flag (F_4), known as "carry-block", not found on most processors. When carry-block is true, the carry is not used during arithmetic operations. This allows simpler code for single-precision operations and some shift operations. The carry-block flag can be saved, restored, or explicitly modified, but is not affected by other operations.

1.3 Memory System and Address Spaces

HERA can address 2^{16} 16-bit words of memory, using the `LOAD` and `STORE` instructions. A HERA CPU typically uses separate memory systems, each with its own address and data buses, for instructions and data (as with the original "Harvard architecture", see Page K-4 of [HP07]).

Chapter 2
Instruction Set

The HERA instruction set is comprised entirely of single-word instructions, each identified by a pattern of bits (in machine language) and a standard abbreviation or short word (in assembly language). In the descriptions below, subscripts are used to identify specific bits, e.g. $b_{15:12}$ refers to bits 15 through 12 inclusive (the first four of the sixteen bits of the instruction). Except for the memory operations `LOAD` and `STORE`, HERA instructions change only registers and flags. Most instructions make use of data from a register, with a few exceptions such as `SETLO`, `INC`, `FSET4`, etc., which use *immediate* operands, i.e., a value that is contained within the bits of the instruction word.

2.1 SETLO and SETHI ($b_{15:13} = 111$)

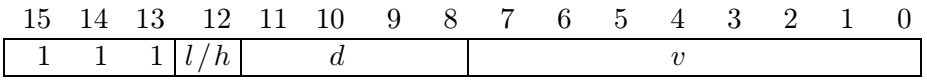
SETLO, and a SETLO/SETHI sequence, are used to place a specific value in a register:

$b_{15:12}$	Mnemonic	Meaning	Notes
1110	`SETLO`(d, v)	$R_d \leftarrow v$	set R_d to signed quantity v
1111	`SETHI`(d, v)	$(R_d)_{15:8} \leftarrow v$	set high 8 bits of R_d

`SETLO` sets R_d to the sign-extended value v, and can thus, by itself, establish a numeric value between -128 and 127. `SETHI` changes the high 8 bits of R_d to v, leaving the lower 8 intact, so a `SETLO`/`SETHI` sequence can be used to establish any 16-bit value.

`SETLO` and `SETHI` do not affect any flags; their actions are not affected by any flag values.

The binary instructions for `SETLO` and `SETHI` identify the instruction ($b_{15:12}$) and destination register ($b_{11:8}$), and include all 8 bits of v ($b_{7:0}$):



2.2 Arithmetic and Bit-wise Instructions ($b_{15:13} = 110$, 101, or 100)

HERA provides the following three-address operations for arithmetic and bit manipulation:

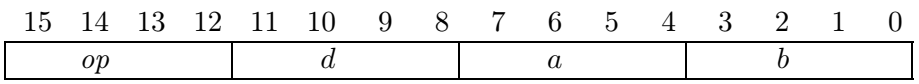
$b_{15:12}$	Mnemonic	Meaning	Notes
1000	`AND`(d, a, b)	$R_d(i) \leftarrow R_a(i) \wedge R_b(i)$	bit-wise logical and
1001	`OR`(d, a, b)	$R_d(i) \leftarrow R_a(i) \vee R_b(i)$	bit-wise logical or
1010	`ADD`(d, a, b)	$R_d \leftarrow R_a + R_b + (c \wedge F_4')$	**a**ddition; use carry unless blocked
1011	`SUB`(d, a, b)	$R_d \leftarrow R_a - R_b - (c' \wedge F_4')$	su**b**traction; use carry unless blocked
1100	`MUL`(d, a, b)	$R_d \leftarrow (R_a * R_b)_{15:0\,[\text{Or } 31:16]}$	multipli**c**ation (details below)
1101	`XOR`(d, a, b)	$R_d \leftarrow R_a \oplus R_b$	bit-wise exclusive or

These operations all set s, to b_{15} of the result, and set z, to true if the result of the operation was zero, otherwise to false. The three bit-wise operations do not affect c or v, but `ADD`, `SUB`, and `MUL` do. The latter set v, to true if there was a signed-arithemetic overflow (i.e., if the signed interpretation of bits 15-0 of their result is not equal to the sum/difference/product (respectively) of the signed interpretations of the operands), or to false if there was no overflow. `ADD` and `MUL` set c, to true if there was a carry (if the sum/product of the unsigned interpretation of the operands is greater than or equal to 2^{16}), and to false otherwise. `SUB` sets c to true if there was *not* a need to borrow from the 2^{16}'s place when subtracting (i.e., if the unsigned interpretation of R_a is greater than that of R_b), and to false if there *was* a need to borrow.

The results of the bit-wise operations do not depend on the values of the flags, but the results of the arithmetic operations do. When carry-block is true, no other flag values matter, and `ADD`, `SUB`, and `MUL` produce bits 15-0 of the sum, difference, or product of the parameters.

When carry-block is false, the arithmetic operations can be used for multiple-precision operations, as they encorporate the value of the carry flag from a presumed prior operation on lower-sigificance words (see Section 4.2 for details). Specifically, when carry-block is false, `ADD` and `SUB` use the incoming value of c as the incoming carry (for `ADD`) or not-borrow (for `SUB`). When carry-block is false, `MUL` can produce a variety of different results, depending on the values of all four other flags. When all are zero, `MUL` produces produces the low word (bits 15-0) of the product of R_a and R_b, exactly as it does when carry-block is true. When only s is true, `MUL` produces the high word (bits 31-16) of the signed interpretations of R_a and R_b. The behavior for the other fourteen possible sets of flag values are currently undefined, but may eventually be defined for multi-word *unsigned* multiplication, or for results that are useful in multiple-precision work.

Three-address arithmetic and bitwise operations are encoded by the op code, then d, a and b.



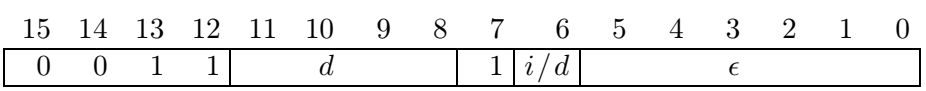
2.3 Increment and Decrement ($b_{15:12} = 0011, b_7 = 1$)

The increment and decrement operations are

$b_{15:12}$	$b_{7:6}$	Mnemonic	Meaning	Notes
0011	10	`INC`(d, δ)	$R_d \leftarrow R_d + \delta$	Increment R_d by δ
0011	11	`DEC`(d, δ)	$R_d \leftarrow R_d - \delta$	Decrement R_d by δ

`INC` and `DEC` update flags as `ADD` and `SUB` would, but always ignore the incoming carry.

The value of b_6 controls whether an increment ($b_6 = 0$) or decrement ($b_6 = 1$) is performed:



Note that the value added or subtracted from R_d is one more than the unsigned quantity given in bits 5:0 (labeled ϵ above) — there is no increment or decrement by zero. By convention, assembly language translators require that the programmer express δ, the quantity to be added or subtracted for `INC` and `DEC`. For example, `INC(r1,6)` produces the machine language instruction $0x3185$, not $0x3186$, to add the constant 6 to R_1.

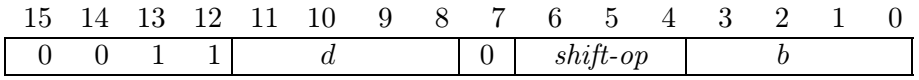
2.4 Shift Instructions ($b_{15:12}=0011, b_{7:4}$ vary)

The HERA shift operations are:

$b_{15:12}$	$b_{7:4}$	Mnemonic	Meaning	Notes
0011	0000	`LSL`(d, b)	$R_d \leftarrow$ shl/rolc (R_b)	Logical shift left, possibly with carry
0011	0001	`LSR`(d, b)	$R_d \leftarrow$ shr/rorc (R_b)	Logical shift right, possibly with carry
0011	0010	`LSL8`(d, b)	$R_d \leftarrow$ shl8 (R_b)	Logical shift left 8 bits
0011	0011	`LSR8`(d, b)	$R_d \leftarrow$ shr8 (R_b)	Logical shift right 8 bits
0011	0100	`ASL`(d, b)	$R_d \leftarrow$ asl/aslc (R_b)	Arithmetic shift left, possibly with carry
0011	0101	`ASR`(d, b)	$R_d \leftarrow$ asr (R_b)	Arithmetic shift right

All shift operations modify s and z as appropriate for the value produced. For one-bit shifts, c becomes the bit shifted out; for `ASL`, v receives the value it would have after `ADD`(d, b, b); otherwise the flag values are not changed by shift instructions.

`LSL`, `LSR`, and `ASL` shift in the value ($c \wedge F_4'$). Thus, when carry-block is false, the logical shift operations correspond to a rotate with carry. `ASR` ignores the incoming carry, always producing a result that is half of the (signed) value. The eight-bit shift operations shift in zeros, regardless of the carry block flag.



2.5 Direct Manipulation of Flags ($b_{15:12}=0011, b_{7:5}=011$)

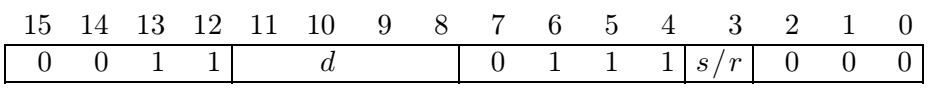
Flag values are automatically set by some operations, but they can also be manipulated directly with the instructions below.

2.5.1 Saving and Restoring Flags

Flags can be collectively saved to, or loaded from, a register.

$b_{15:12}$	$b_{7:4}$	b_3	Mnemonic	Meaning	Notes
0011	0111	0	`SAVEF`(d)	$R_d \leftarrow F$	Save flags to R_d
0011	0111	1	`RSTRF`(d)	$F \leftarrow R_d$	Restore flags from R_d

Note that the flags are saved in bits 4-0 of R_d. Instruction bit b_3 controls whether flags are saved ($b_3=0$) or restored ($b_3=1$):



2.5.2 Setting Flags

Sets of flags can be explicitly turned on (set to true) or off (set to false) with `FON` or `FOFF`. It is also possible to set all flags (or all but carry-block) via `FSET5` (or `FSET4`).

$b_{15:12}$	$b_{11:9}$	$b_{7:4}$	Mnemonic	Meaning	Notes
0011	000	0110	`FON`(v)	$F \leftarrow F \vee v$	Set to true any flags for which v is true
0011	100	0110	`FOFF`(v)	$F \leftarrow F \wedge v'$	Set to false any flags for which v is true
0011	010	0110	`FSET5`(v)	$F \leftarrow v$	Set all flags to have the values of v
0011	110	0110	`FSET4`(v)	$F_{3:0} \leftarrow v$	Set flags other than carry-block

The binary representations for these instructions combine b_8 and $b_{3:0}$ to represent v:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	*flag-op*			v_4	0	1	1	0	v_3	v_2	v_1	v_0

For example, `FON(0x15)` (i.e., binary $0x$`3165`) sets the carry block (F_4, which is $0x10$ in the flag register) and v and s ($0x04$ and $0x01$) to true, but leaves c and z unchanged; `FOFF(0x0a)` (binary $0x$`386A`), which clears c and z, leaving F_4, v, and s; `FSET4(0x5)` (binary $0x$`3C65`) sets v and s to true, and c and z to false (carry-block is unchanged by `FSET4` but affected by `FSET5`).

2.6 Memory Instructions ($b_{15:14} = 01$)

The `LOAD` and `STORE` instructions move data between registers and memory.

$b_{15:13}$	Mnemonic	Meaning	Notes
010	`LOAD`(d, o,b)	$R_d \leftarrow M[R_b + o]$	Load memory cell into R_d.
011	`STORE`(d, o,b)	$M[R_b + o] \leftarrow R_d$	Store value of R_d into memory.

No flag is modified or used during a `STORE` instruction; `LOAD` modifies s and z and leaves other flags unchanged. The binary format for these instructions is

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	l/s	o_4		d			o_3	o_2	o_1	o_0		b		

where b_{13} is 0 for a `LOAD` operation and 1 for a `STORE`, and data is transferred between R_d and memory cell $R_b + o$, where o is a 5-bit *unsigned* number (0..31) constructed from b_{12} followed by $b_{7:4}$. For example, `LOAD(r7, 0x13,r2)` (binary $0x$`5732`) loads $M[R_2 + 0x13]$ into R_7.

2.7 Control-Flow ($b_{15:14} = 00$)

Control-flow instructions include conditional branches, unconditional branches (also called "jump"), and transfers among functions and coroutines. Op codes are also set aside for software functions associated with interrupt processing, though this behavior is not fully specified.

2.7.1 Branches, including jumps ($b_{15:13} = 000$)

HERA provides the following branch instructions to vary the program counter's normal progress through the instruction addresses (i.e., the usual update $PC \leftarrow PC + 1$). Branch instructions do not change any flags.

Register-mode branches set PC to R_b, and relative branches set PC to $PC + o$. Thus, the unconditional relative branch instruction includes the special cases `HALT` (when $o = 0$) and `NOP` ("no operation", when $o = 1$). For a conditional branch, this adjustment only occurs if the stated flag expression is true; otherwise, the PC changes to the usual $PC + 1$.

$b_{15:12}$	$b_{11:8}$	Mnemonic	Meaning
0001/0	0000	BR(b)/BRR(o)	Always set $PC \leftarrow R_b$ (for BR) or $PC \leftarrow PC + o$ (for BRR)
0001/0	0001		(*unused*)
0001/0	0010	BL(b)/BLR(o)	Branch if signed result <0, i.e., set PC if $(s \oplus v) = \text{True}$
0001/0	0011	BGE(b)/BGER(o)	Branch if signed result ⩾0, i.e., $(s \oplus v)'$
0001/0	0100	BLE(b)/BLER(o)	Branch if signed result ⩽0, i.e., $((s \oplus v) \vee z)$
0001/0	0101	BG(b)/BGR(o)	Branch if signed result >0, i.e., $((s \oplus v) \vee z)'$
0001/0	0110	BULE(b)/BULER(o)	Branch if unsigned result ⩽0, i.e., $(c' \vee z)$
0001/0	0111	BUG(b)/BUGR(o)	Branch if unsigned result >0, i.e., $(c' \vee z)'$
0001/0	1000	BZ(b)/BZR(o)	Branch if zero, i.e., z, i.e., if equal operands (of SUB/CMP)
0001/0	1001	BNZ(b)/BNZR(o)	Branch if not zero, i.e., z', i.e., if operands not equal
0001/0	1010	BC(b)/BCR(o)	Branch if carry, i.e., c , i.e., if unsigned result ⩾0
0001/0	1011	BNC(b)/BNCR(o)	Branch if not carry, i.e., c', i.e., if unsigned <0
0001/0	1100	BS(b)/BSR(o)	Branch if sign, i.e., s, i.e., signed result was negative
0001/0	1101	BNS(b)/BNSR(o)	Branch if not sign, i.e., s'
0001/0	1110	BV(b)/BVR(o)	Branch if overflow, i.e., v
0001/0	1111	BNV(b)/BNVR(o)	Branch if not overflow, i.e., v'

Branches, both conditional and unconditional, are indicated by $b_{15:13} = 000$. For both, $b_{11:8}$ indicate the condition under which the branch is to be taken, unconditional branch is indicated by $b_{11:8} = 0$; see Section 9-8 of Mano's "Computer Engineering: Hardware Design" [Man88] for details of flag usage. Register-mode branches are indicated by an appended "R" in the assembly-language name, and by $b_{12}=1$ (vs. $b_{12}=0$ for relative branches) in machine code.

The format for register-mode branch instructions is

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	*cond*				0	0	0	0	*b*			

The format for relative branch instructions is

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	*cond*				*o*							

where the 8-bit *signed* quantity o gives the amount to adjust the PC (-128...127).

2.7.2 Function call and return ($b_{15:9} = 0010\,000$)

HERA provides a single machine-language mechanism to perform both call and return. To ease debugging, and for historical reasons, CALL and RETURN are expressed with two different op codes, though they have the same hardware semantics and can be used to implement arbitrary transfer among co-routines as well as functions.

$b_{15:8}$	Mnemonic	Comments
0010 0000	CALL(a, b)	Call function at address R_b, with new stack frame starting at R_a
0010 0001	RETURN(a, b)	Return, if return address and caller's FP are in R_b and R_a

Both of these instructions are achieved via $PC \leftarrow R_b$, $R_b \leftarrow PC + 1$, $FP \leftarrow R_a$, $R_a \leftarrow FP$. Neither changes any flags.

Since the action of these instructions is the same, hardware can ignore b_8 of the instruction.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	*0/1*	*a*				*b*			

The difference between `CALL` and `RETURN` is thus one of *usage*, not implementation. To perform a `CALL` to a function f, R_b should be f's starting address, R_a should be the base (minimum) address for the stack frame to be used for this call to f, and the stack pointer (R_{15}, a.k.a. SP) should be sufficiently far above R_a to accomodate f's initial stack frame. The `CALL` instruction itself would then set $PC \leftarrow \text{start}(f)$ and $FP \leftarrow \text{frameBase}(f)$, and save the return address in R_b and the caller's frame pointer in R_a. The corresponding `RETURN` would put that return address back into PC, and the caller's frame pointer back into FP, as long as calling and called functions have consistent conventions, for example always using R_{12} as R_a, and R_{13} as R_b, and assuming the values have not been overwritten by the body of the function.

See Chapter 7 for more information about conventions for function calls. The use of the HERA `CALL` (or `RETURN`) instruction for other transfers of control, e.g., for iterators, generators, or other co-routines, is outside the scope of this document.

2.7.3 Codes Reserved for Interrupt Processing ($b_{15:9} = 0010\,001$)

The precise working of interrupts, and of input and output, involve interplay between hardware signals and program instructions. The co-design of these mechanisms is specifically left as an exercise, e.g. for students in Haverford's CMSC 356: Concurrency and Co-Design in Operating Systems. However, the names and operation codes below are reserved for specific purposes, and assemblers and other tools should recognize/support them to the degree possible:

b				Mnemonic	Comments
0010	0010	0000	i	`SWI`(i)	Software interrupt $\#i$
0010	0011	0000	0000	`RTI`()	Return from interrupt

`SWI` allows software to simulate an interrupt, though of course this "interrupt" is synchonized with the excecution of the `SWI` instruction, rather than being triggered by some external event such as a user typing on a keyboard. `RTI` returns from an interrupt, whether triggered by `SWI` or by a hardware signal.

Other operation codes with $b_{15:12} = 0010$ (and $b_{11:10} \neq 00$) are completely unspecified.

Chapter 3
Assembly-Language Features

Assemblers and other tools for low-level code often recognize a number of statements beyond the set of instructions provided in hardware. Typically, they provide *pseudo-operations* that combine several hardware instructions for convenience; they let programmers enter a program's *data* as well as well as its instructions; they let programmers define and use *labels*, i.e., symbolic names for constants, including memory locations; and they let programmers define their *macros* to extend the notation used in their programs.

3.1 Pseudo-Operations and the Temporary Register

As with real RISC architectures, the HERA instruction set defines a core of features that can each serve a variety of purposes, avoiding the need for hardware implementations of some features found in non-RISC architectures. For example, HERA has no "move" instruction to move a value from one register to another, as a bit-wise or with R_0 will do this job; HERA has no bit-wise not operation, as an exclusive-or with the constant $0x$ffff will do this; HERA has no instruction to test if one register's value is less than another's, as a subtraction that discards its numeric result (by directing it into R_0) will do so.

To provide standardization and convenience, a number of these common idioms are normally pre-defined in assemblers. Thus, an assembler will translate both actual hardware instructions, turning `FSET(0x08) SUB(5,2,3)` into $0x$`3068` $0x$`B523`, and the pseudo-operations below, turning `CMP(2,3)` into $0x$`3068` $0x$`B023`. (Note that some assemblers may also provide an option to skip the step of setting the carry before subtraction in the `CMP` pseudo-operation, if they can ensure that this shorter definition will only be used when the carry-block is set, but these details are beyond the scope of this document.)

The stardard HERA pseudo-operations for arithmetic and flag operations are shown below:

Mnemonic	Definition	Notes
`SET`(d, v)	`SETLO`(d, v `&` $0x$ff); `SETHI`(d, $v \gg 8$)	$R_d \leftarrow v$ (set R_d to 16-bit value v)
`SETRF`(d, v)	`SET`(d, v); `FLAGS`(d)	$R_d \leftarrow v$ (set R_d and flags for $v+0$)
`MOVE`(a,b)	`OR`(a, b,R_0)	$R_a \leftarrow R_b$
`CMP`(a, b)	`CON`( ); `SUB`(R_0, a, b)	Set flags for $a - b$
`NEG`(d, b)	`CON`( ); `SUB`(d, R_0, b)	Set $R_d \leftarrow -R_b$
`NOT`(d, b)	`SET`(R_{11}, $0x$ ffff); `XOR`(d, R_{11}, b)	Set R_d to the Bit-wise complement of R_b
`CON`( )	`FON`($0x$08)	Turn on the carry flag
`COFF`( )	`FOFF`($0x$08)	Turn off the carry flag
`CBON`( )	`FON`($0x$10)	Turn on the carry-block flag
`CCBOFF`( )	`FOFF`($0x$18)	Turn off the carry and carry-block flags
`FLAGS`(a)	`COFF`( ); `ADD`(R_0, a, R_0)	Set flags for R_a
`OPCODE`(n)	n	Machine language op n

Note that R_{11} is used as a "temporary register" to hold a value during the steps of the `NOT` pseudo-operation. Assemblers typically also define `Rt` as a synonym for `R11`, to encourage programmers to use this same temporary register (as long as they do not expect it to retain a value while a `NOT` pseudo-operation is performed or during an assembler-generated register-mode branch to a labelled statement, as discussed below).

The `OPCODE` pseudo-operation is typically used to translate non-standard extensions of the HERA instruction set.

3.2 Instruction Labels and Branching Pseudo-Operations

Proper use of branching instructions is among the most tedious and error-prone elements of direct machine-language programming. When writing a program, the programmer must count the number of instructions to be skipped in a relative branch, or find the absolute program address to be reached in a register-mode branch. When editing a program in a way that inserts or removes instructions, all relative branches across the edit point, and all register-mode branches after that point, must be updated.

Assemblers ease this burden greatly by letting programmers identify program instructions with labels, and then refer to those labels in branch instructions. The correct numbers are then computed as the code is assembled.

The standard HERA pseudo-operations for branching are shown below:

Mnemonic	Definition	Notes
`LABEL`(L)	*(no machine language generated)*	Define L = next instruction's address
`BR`(L)	`SET`(R_{11}, address(L)); `BR`(R_{11})	Branch to label L, using R_{11}
`BRR`(L)	`BRR`(distanceTo(L))	Branch (relative) to label L
`BG`(L), `BC`(L)...	...	(conditional branches use R_{11})
`BGR`(L), `BCR`(L)...	...	(conditional relative branches)
`NOP`( )	`BRR`(1)	Do nothing ("No operation")
`HALT`( )	`BRR`(0)	Halt the program
`CALL`(a, L)	`SET`(R_{13}, address(L)); `CALL`(R_a, R_{13})	Call a function, using R_{13} for address

A HERA label is a sequence of letters, numerals, and underscores that does not start with a numeral (i.e., any legal C++ identifier), and identifies the instruction that follows it.

Note that R_{11} is once again used as a "temporary register" that holds a value within the steps of a (register-mode branch) pseudo-operation. Since the registers used in a `CALL` retain important values after the execution of the call instruction, the pseudo-op uses R_{13} rather than R_{11} to hold the address; by convention, the first operand of `CALL` is R_{12}, and assemblers may reject code that does not follow this convention, or the matching convention of using R_{12} and R_{13} as the operands of all `RETURN` instructions. See Chapter 7 for details of idiomatic usage of `CALL` and `RETURN`.

Some assemblers and other tools may support branches only to labels (rather than specific addresses or offsets).

3.3 Data Statements and Data Labels

Programs contain data as well as code. HERA assembly language provides a set of standard data statements to pre-load integer and string values into data memory, to allocate uninitialized memory, to name data-memory locations with data labels, and to provide symbolic names for constant values.

Mnemonic	Meaning
`CONSTANT`(N, v)	Define the name N to have the value v
`DLABEL`(L)	Define L = the next data statement's address
`INTEGER`(i)	Put i in the current data-memory cell
`LP_STRING`("s")	Put length-prefixed string s in data memory
`DSKIP`(n)	Skip n cells of data memory

The `CONSTANT` statement defines a symbolic name for a value, for use in later data statements or instructions. Like `LABEL` and `DLABEL`, it does not produce any machine language by itself, but instead guides the assembler in the translation of later parts of the program.

Data labels follow the same naming rules as instruction labels, but are used to label the memory cells allocated with the following `INTEGER`, `LP_STRING`, and `DSKIP` statements.Strings must be entered as printable ASCII characters and spaces, with backslash (\) used only for the sequences `\t`, `\n`, `\\`, `\"`, `\x`*hh*, and `\u`*hhhh* (for hexidecimal digits *h*). The escape sequences have the meanings they would have in C/C++ strings, but note that `LP_STRING` stores s in data memory as an integer count of the number of characters followed by the characters, rather than as a C-style null-terminated sequence.

Note that the HERA-C simulator, HERA assemblers, and other assembly-language tools may require that all data statements and data labels precede all instructions and instruction labels.

Part II

Idioms

Chapter 4
Idioms for Operating on Scalar Values

High-level languages typically allow complicated expressions involving both variables and literal values, and allow assignment of values to variables, for example `a := 7*(b+c)+4*d` and `e := d-c` (for integer variables `a-e`), or `ch>='A' and ch<='Z'` (for a character variable `ch`). In HERA, as with most RISC architectures, arithmetic and comparisons are performed on registers, so we must make registers take on the roles of variables and literal values.

4.1 Single-Precision Values, Arithmetic, and Comparisons

When every value can fit in a 16-bit register, we place each literal value or variable in a single register; this is known as *single-precision* arithmetic, since we use a single register for each value. In single-precision arithmetic, each addition, subtraction, or multiplication can be performed with a single HERA operation, without requiring the use of a flag to carry information between registers. Each use of a variable is equivalent to the use of the associated register as an operand; each definition of a variable corresponds to the use of the associated register as a destination. Programs that contain *only* single-precision arithmetic typically begin by turning the carry-block flag on via `CBON()`.

The main challenges in writing single-precision arithmetic in HERA are the potential to run out of registers (see Chapter 6) and the need to put each intermediate result and literal, as well as each variable, in a register. Sometimes R_t (the temporary register, R_{11}) or the register for the variable being defined can serve these purposes. So, for example, we might think of the expression `a := 7*(b+c)+4*d` as the sequence of binary operations `a:=b+c; Rt:=7; a:=Rt*a; Rt:=4; Rt:=Rt*d`. Figure 4.1 shows the HERA equivalent of `a := 7*(b+c)+4*d; e := d-c`, assuming we associate `a` with R_1, `b` with R_2, etc.. Note the use of `SETLO` for the literals 7 and 4; `SET` could also be used, though a single `SETLO` should require less time and consume less energy than the

```
// Set R1 to the single-precision sum of 7*(R2+R3)+4*R4, R5 to R4-R3

CBON()     // Turn on carry-block, disabling carry flag, for  single-precision
  //  ...           (possibly more instructions other than CBOFF)

// start by computing 7*(R2+R3)
ADD(R1, R2,R3) // R1 = R2 + R3, regardless of incoming carry flag
SETLO(Rt, 7)   //  Put 7 in a register so that we can multiply by 7
MUL(R1, Rt,R1) // With CB on, MUL computes the low 16-bits of the product
// next, find 4*R4 and add it to the above
SETLO(Rt, 4)
MUL(Rt, Rt,R4) // Rt is now 4*R4
ADD(R1, R1,Rt) // R1 = R1 (i.e. 7*(R2+R3)) + Rt (i.e., 4*R4)

// last step: R5=R4-R3
SUB(R5, R4,R3) // R5 = R4 - R3, regardless of carry flag
```

Figure 4.1. Single-Precision Arithmetic.

`SETLO`/`SETHI` sequence produced by `SET`. `SET` has the advantage of working for all single-precision values, rather than just the range $-128 \ldots 127$. The HERA machine language for Figure 4.1 is `3160 a123 eb07 c1b1 eb04 cbb4 a11b b543` (here and below, machine language is written in hexidecimal without the preceding `0x`).

Arithmetic operations that are not included in the HERA instruction set, such as division, remainder, exponentiation, etc., are typically handled by calling a function written in HERA (possibly as part of a library of functions; the Tiger standard library for HERA includes single-precision integer division and remainder functions). The use and creation of HERA functions is discussed in Chapter 7.

Note that there may be many HERA equivalents for a single high-level language program, just as there are multiple ways to express a given algorithm in a high-level language (such as the five-step sequence of assignments discussed as an equivalent to `a := 7*(b+c)+4*d` above). As noted above, `SET(Rt, 4)` and `SETLO(Rt, 4)` produce the same result. Additionally, Figure 4.1 uses the sequence `SETLO(Rt, 4) MUL(Rt, Rt,R4)` to multiply the value of R_4 by four; the sequence `ASL(Rt, R4) ASL(Rt, Rt)` would produce the same result (apart from flags, which we're ignoring in single-precision). In some cases, a clever sequence can save significant programming effort, for example if we had needed to *divide* R_4 by 4, we could have written `ASR(Rt, R4) ASR(Rt, Rt)` with much less effort than we would have expended in writing a general-purpose division function. Such distinctions among different ways of producing the a result can be of vital importance when programming performance-critical software or writing an optimizing compiler, but the exact impact of a given change sometimes depends on the hardware design of the HERA chip, and so this topic will be largely ignored hereafter.

HERA assemblers accept single-quoted characters (i.e., characters surrounded by apostrophes, such as `'?'`) according to the rules given in Section 3.3 for details of strings. For example, the assembly language `SETLO(R3, 'X')` would be translated into `e358`, since the ASCII and Unicode [Uni12] encoding systems give 'X' the same binary pattern that would be used for the integer 88 ($0x58$). The assembly-language `SETLO(R3, 88)` and `SETLO(R3, 0x58)` also produce `e358`.

Comparisons are usually performed with the `CMP` pseudo-operation; literal character values are often placed in R_t or a register that does not currently hold an important value (as shown with literal integer values above). So, to check if a value in R_3 is $\geqslant$ the value for 'A', we could use the sequence `SETLO(Rt, 'A') CMP(R3, Rt)`, after which the flag would set according to the result of $R_3 - R_t$ (i.e., $R_3 \geqslant$ 'A', the subtraction would be nonnegative, so s would be off). Since `CMP` changes flags but not register values, it is typically followed by a branch operation, as discussed in Chapter 5.

4.2 Double-Precision or Mixed-Precision Arithmetic

To manipulate values that cannot fit in a single 16-bit register, we can use multiple registers to record a single value. For example, 32-bit numeric values can be recorded in pairs of registers; this is known as *double-precision* computation. Figure 4.2 illustrates double-precision addition, using R_1 and R_2 together to represent a 32-bit value, with R_1 holding the higher-order (leftmost 16) bits and R_2 the lower-order (rightmost 16) bits. The program gives the $[R_1\ R_2]$ pair the sum of three other 32-bit values.

Note that each double-precision addition step consists of a pair of HERA `ADD` instructions, the second of which will respect the value of the carry flag that was produced in the first. This is analogous to the fact that we must respect the fact that there may or may not be a carry when we manually sum two digit numbers, e.g., 18 + 24 or 31 + 11. Note that these pairs of `ADD` instructions are preceded by a step to explicitly turn the carry flag off, to avoid inappropriately adding 1 to our result if c happened to have been set prior to starting the sequence.

A sequence of SUB instructions will similarly use c to ensure that borrowing takes place appropriately. Note that, since c is defined as the *absence* of a borrow, c must be turned *on* before starting a sequence of one or more SUB instructions (unless, of course, carry-block is on). The last three lines of Figure 4.2 illustrate double-precision subtraction. Without the preceding CON instruction to set c to true, the c flag would presumably be false, since we expect the sum computed in the previous steps to fit in 32 bits (if we were not confident of this fact, we could use higher precision, or use a branch instruction to test c and branch to error-handling code if our expectation did not hold). Thus, in the absence of a step to explicitly set c to true and indicate the absence of a prior need to borrow, the SUB(r8, r8,r2) would set $R_8 = R_8 - R_2 - 1$ rather than the hoped-for $R_8 = R_8 - R_2$. While the convention "carry indicates the *absence* of borrow" may seem odd to novice assembly-language programmers, it simplifies some hardware components, and is thus found on many microprocessors.

Multiplication of double- (or higher) precision values is left as a challenge for the reader. The HERA machine language for Figure 4.2 is `3968 3868 a246 a135 3868 eb40 fb42 a22b eb0f a11b 3068 b882 b771`.

4.3 Higher Precision and Non-Integer Numeric Values

Larger groups of registers can, of course, be used for higher precision, though when registers cannot hold all bits of all values, some data must be stored in main memory (see Chapter 6). Non-integer numeric types typically found in high-level languages, such as floating-point approximations of real numbers, rational numbers, complex numbers, vectors, matrices, etc., are usually implemented in HERA as user-defined data types. Values of user-defined types can be seen as sets of bits stored in registers or main memory, for which all operations are performed via calls to

```
// Make [R1 R2] the (double-precision) sum of [R3 R4] + [R5 R6] + 1000000;
// finally, update [R7 R8] by subtracting the above total from it

CCBOFF()       // Enable use of carry flag for multiple-precision arithmetic
  // ...           (possibly other instructions that may set or clear the carry)

// start by finding [R3 R4] + [R5 R6]
COFF()         // Start with carry-in=0 for least-significant word
ADD(R2, R4,R6) // R2 = R4+R6, carry set if necessary
ADD(R1, R3,R5) // R1 = R3+R5 (plus carry, if set by R4+R6)
// now add 1000000 to the above
COFF()         // Start with carry-in=0 for least-significant word
SET(Rt, 16960) //  16960 is the lower 16 bits of 1000000, i.e., 1000000 %65536
ADD(R2, R2,Rt) // R2 = R2+Rt (i.e. R4+R6+16960), carry set if necessary
SETLO(Rt, 15)  //  15    is the higer 16 bits of 1000000, i.e., 1000000//65536
ADD(R1, R1,Rt) // R1 = R1+Rt (plus carry, if set), i.e., high word of the sum

// next calculation: subtract the above from [R7 R8]
CON()          // Make sure carry-in=1 for subtraction without borrow
SUB(R8, R8,R2) // Set low  word to R8-R2 (leaves carry-flag=0 if it borrowed)
SUB(R7, R7,R1) // Set high word to R7-R1 (-1 if incoming carry was 0)
```

Figure 4.2. Double-Precision Arithmetic.

functions that encapsulate the details of how the bits relate to abstract values. This is essentially like a definition of a class in which the data fields are a collection of bits (or characters or integers); it can also be seen as a generalization of the implementation of integer division and remainder in HERA library functions.

Chapter 5
Local Control Flow

High-level languages provide a variety of keywords, such as `if`/`then`/`else`, `while` loops, `for` loops, etc., for varying the usual line-by-line order of program execution within each function (or procedure, coroutine, or method). HERA provides a variety of branch operations that can be used to create patterns equivalent to `if`/`then`/`else`, `while`, `for`, and many other appropriate and inappropriate forms of control flow within a function (with `CALL` and `RETURN` handling transfer between functions, etc., as will be discussed in Chapter 7). After significant unpleasant experience with "unstructured control flow", assembly-language programmers now often choose to think about local control flow in high-level terms. Specifically, they typically start coding by writing high-level constructs (`if/then/else`, etc.) as comments, and then write the assembly-language among these comments. Thus, the only unfamiliar task is the choice of branching instructions that corrospond to the high-level structure.

5.1 Choosing Appropriate Branch Instructions

Register-mode branches such as `BR`/`BGE`/`BZ` differ from the corresponding relative branches `BRR`/`BGER`/`BZR` in that the former require that the destination address be placed in a register (R_t, by convention, in assembler-generated code), and the latter require that the destination not be too far from the branch itself. So, for example, after the sequence `SET(Rt, 0x0174) BR(Rt)` (or, equivalently, `eb01 fb74 100d` in machine language), the processor will execute whatever instruction is in instruction memory cell $0x0174$, then $0x0175$, then $0x0176$, etc. (unless $0x0174$ or $0x0175$ contains a branch, call, or return instruction). A single `BRR(4)` instruction (`0004` in machine language) in instruction address $0x170$ would also branch to $0x0174$, as would a `BRR(-4)` (`00fc` in machine language) in instruction address $0x0178$, but there is no way to get from instruction address $0x3944$ to $0x170$ with a relative-mode branch. Register-mode branches are thus more general, in that they are always correct as long as no important information has been left in R_t, but relative branches may execute more efficiently when they are legal.

Conditional branches such as `BZ`/`BZR` (branch on zero) and `BGE`/`BGER` (branch on greater-than-or-equal) transfer only under a specific condition, otherwise going on to the next instruction; they thus correspond to branching points in the flow of control, such as after the test of an `if` or `while`, or after a `for` loop determines whether or not another iteration is needed. Unconditional branches (`BR`/`BRR`) always transfer control to the destination instruction, as would be done when code in a high-level-language completes statements of the "`then`" part of an `if`/`then`/`else` and unconditionally skips over the statements controlled by the `else`, or when execution of a `while` loop "goes back up" to test the condition again after executing the statements of the loop body.

5.2 Combining Tests and Branches

Conditional control flow in HERA is typically expressed via a combination of arithmetic or comparison instructions, which set flags, and conditional branches, whose actions will be controlled by those flags. Figure 5.1 shows a program that uses `CMP(R1,R0)` to adjust the flags as they would be for $R_1 - 0$, and then `BGER` to branch if the flags indicate a non-negative result for subtraction of signed integer quantities (i.e., if $(s \oplus v)$ is false, indicating a non-negative value without an overflow or an apparently negative value in the presence of overflow). Thus, the negation of the value in R_1 is skipped if R_1 is already positive. Figure 5.1 illustrates a style of indenting branches, calls, and labels less than other code, to draw attention to the program structure, though this is not required for the code to execute correctly.

An assembler would typically produce `3160 e1b6 3068 b010 0303 3068 b101 3111` for this program. If we had used a `BGE` instead of `BGER`, and it were placing instructions into addresses starting at $0x0200$, it would typically use `SET` to put $0x0209$ (the address of the `LSR` instruction labelled `SKIP_NEGATION`) into R_{11}, producing `3160 e1b6 3068 b010 eb09 fb02 130b 3068 b101 3111` instead.

```
// Make R1 = (abs(R1))/2 (using R1=-74, for example). In other words:
//   if     R1 < 0
//   then   R1 = -R1
//   endif
//   R1 = R1/2

  CBON()              // Use single-precision
  SETLO(R1, 0xB6)     // Try -74 for this example

// if R1 < 0          // i.e., skip the "then" part if R1>=0
  CMP(R1,R0)          // set flags for R1-0, i.e., SUB(R0, R1,R0)
BGER(SKIP_NEGATION)   // if flags show result >= 0, skip over SUB
// then R1=-R1
  NEG(R1, R1)         // negate R1, i.e., SUB(R1, R0,R1)
// endif
LABEL(SKIP_NEGATION)
// R1=R1/2
  LSR(R1, R1)         // divide by 2 via logical shift right
```

Figure 5.1. Control Flow and HERA Branch Instructions.

Chapter 6
Bigger Data

When a program needs to process more data than can fit in registers, some data must be placed in the main memory system. HERA follows the traditional view of the computer's memory system—essentially, it views memory as one large array of values, in which each element is identified by an integer index known as the *address* of the element. Assembly-language `LOAD` and `STORE` instructions provide a common foundation that can be used to handle a variety of situations in which not all information fits in registers, including high-level language features such as arrays, strings, record/class-type objects, "bignum" numbers (i.e., integers potentially having values too large for even double-precision), separate variables for each function, and large numbers of variables within one function.

HERA's `LOAD` instruction retrieves a value from a memory cell and places it into a register; the value in a register can be stored into a memory cell via the `STORE` instruction. The address of the memory cell used in a `LOAD` or `STORE` is computed by adding a small constant offset (often zero) to a "base address" given in a register. For example, to retrieve the value of memory cell 6 into register 3, we could put 6 into a register (e.g. `R1`) and then use a `LOAD` with offset zero (e.g., `SET(R1, 6) LOAD(R3, 0,R1)`). Conversely, to store the value in `R7` into memory cell 6, we could use `SET(R1, 6) STORE(R7, 0,R1)`.

HERA's data statements (see Section 3.3 and examples below) can be used to place data in memory before a program starts. An assembler would convert a program with data statements into a pair of sequences of values, one for the data memory, and one for the instruction memory. The machine language values for examples are based on the assumption that the assembler places the first data statement in cell $0x$C001 of the data memory, and the first instruction in cell 0 of the instruction memory, as is done by the HERA-C system and Hassem assembler.

The remainder of this chapter uses `LOAD`, `STORE`, and data statements to illustrate many of the situations for which data are typically stored in main memory: values of some types, such as arrays, strings, and user-defined classes, may be too large to fit in the available registers, and even for small types such as single-precision integers or characters, the number of variables may exceed the number of available registers. Main memory is also used to provide separate local variables for each function, but techniques for creating HERA functions are subtle enough to warrant their own chapter. Memory is also used to implement the free-store heap, in languages that use this mechanism for potentially-long-lived objects created during program execution, but alogrithms for managing a free-store heap are beyond the scope of this document.

6.1 Global Variables Stored in Main Memory

The assembler will automatically allocate memory in response to the `DSKIP` and `INTEGER` data statements, with the latter initializing the memory to a single-precision integer (or character) value. Memory allocated in this way can be identified by preceding it with a `DLABEL`, which can then be used as the operand of a `SET` operation that places its address in a register (which is then typically used as the address operand of a `LOAD` or `STORE`).

Figure 6.1 illustates two idioms for allocating and using variables in main memory. For the first (used for `Y:=X+5`), each allocated object is given its own `DLABEL`, and that name is `SET` into a register before the variable is used (via `LOAD`) or set (via `STORE`). For the second, a single `DLABEL` precedes the set of objects, and then the offset operand of the `LOAD` and `STORE` is used to identify the memory locations we think of as `X` (at offset 0 from `Variables`), `Y` (offset 1), and `Z` (offset 2). Note that offsets are restricted to the range 0....31; programs with very large sets of variables can employ multiple labels or combine some offsets via explicit `ADD` instructions.

The assembler would both produce a sequence of instructions `(3160 eb01 fbc0 410b 3184 eb02 fbc0 610b e101 f1c0 4201 4311 a333 a223 4321 b223 6201 0000` and a set of data-memory initializations (putting `000c`, i.e., 12, into cell $0x$C001, and `0004` into $0x$C003) for the program shown in Figure 6.1. Note that the Hassem assembler, the HERA-C system, and other tools for HERA may require that all data statements precede any instructions.

```
// Given (single-precision) integer variables X, Y, and Z,
//  with X initialized to 12 and Z initialized to 4,
//  execute "Y:=X+5; X:=X+2Y-Z".

// First, data statements allocating and perhaps initializing X, Y, Z
  DLABEL(Variables)
  DLABEL(X)
    INTEGER(12) // Initial value of X
  DLABEL(Y)
    DSKIP(1)    // Uninitialized space for 1 variable (Y)
  DLABEL(Z)
    INTEGER(4)  // Initial value of Z

// And now, the program ... first, select single-precision
    CBON()

// Y:=X+5, using the "specific labels for each variable" idiom
    SET(Rt, X)          // Rt is now the _address_ of X
    LOAD(R1, 0,Rt)      // R1 is now the _value_ of X
    INC(R1, 5)          // R1 is now X+5
    SET(Rt, Y)          // Rt is now the addres of Y
    STORE(R1, 0,Rt)     // Store that X+5 in address for Y

// X:=X+2Y-Z, using the "specific offsets from one label" idiom
    SET(R1, Variables)
    LOAD(R2, 0,R1)      // R2 is now X (0 past "Variables")
    LOAD(R3, 1,R1)      // R3 is now Y
    ADD(R3, R3,R3)      // R3 := R3+R3, now R3 is 2Y
    ADD(R2, R2,R3)      // R2 := R2+R3, so  R2 is X+2Y
    LOAD(R3, 2,R1)      // R3 is now Z (and R2 X+2Y)
    SUB(R2, R2,R3)      // R2 := R2-R3, so  R2 is X+2Y-Z
    STORE(R2, 0,R1)     // Store R2 as new X value, so X:=X+2Y-Z

  HALT()        // Program is now done
```

Figure 6.1. Integer Variables Stored in Memory

6.2 Arrays and Address Arithmetic

The assembly language equivalent of an array variable is a sequence of consecutive memory cells that contain the values of the array, and for which the address of the initial element is known. Figure 6.2 shows an example with two such arrays, following the convention that the first value is preceded by the array size.

Using this convention, we can find the address of a particular element by multiplying the array index by the element size and adding a "base address" for the array. If the array contains single-precision values, we can omit the multiplication by one, of course, and find the base address via a `DLABEL` before the memory cell for the size (if we like to think about array indices starting at one; if we like to count subscripts from zero, a `DLABEL` can be placed before the initial value in the array). The lines involving `Rt` in Figure 6.2 use this approach to place the value 11 in the fifth entry of `ArrayOfPrimes` (the value 11 could, of course, have been placed there in the initialization step instead).

This addition of base address and index could be done for every reference to an array, though for many programs this can be greatly simplified. For example, for a program that goes through all elements of an array in order, a register can be used to identify the address of the "current" element, and this address simply incremented to move to the next element. Figure 6.2's creation of `ArrayOfSevenSquaredOfPrimes` shows an example of this approach, in which `R1` holds the address of the current element about to be read, and `R2` the address of the element about to be written.

The machine-language version of Figure 6.2 is `3160 eb01 fbc0 e105 abb1 e10b 610b e101 f1c0 e209 f2c0 4301 6302 33c0 0207 3180 3280 4401 c444 6402 00f9 0000`; the data memory will be initialized with `0007 0002 0003 0005 0007` in locations $0x$C001...$0x$C005, and `000D 0011` in $0x$C007 and $0x$C008.

Multi-dimensional arrays are represented in a variety of ways in high-level languages. For example, a three by seven array of single-precision values could be handled as an array of three elements of seven single-precision values (i.e., a set of 21 values in consecutive memory cells); or it could be handled as an array of three single-precision values, each of which provides the memory location of an array of size seven; or it could be handled in a variety of other ways if we needed to, for example, efficiently select either rows or columns from the two-dimensional structure. The advantages and disadvantages of these approaches are outside the scope of this document.

```
// Given a length-prefixed array of very small (below sqrt(2**16)) primes,
//  compute the square of each one,
//  placing results in an array of single-precision values.


// ArrayOfSevenPrimes is initialized to [2, 3, 5, 7, -, 13, 17]
  DLABEL(ArrayOfSevenPrimes) // Note that 11 is missing in the array below
    INTEGER(7) // array size given first, by convention
    INTEGER(2) INTEGER(3) INTEGER(5) INTEGER(7) DSKIP(1) INTEGER(13) INTEGER(17)
// ArrayOfSevenSquaredPrimes is 8 uninitialized spaces
  DLABEL(ArrayOfSevenSquaredPrimes)
    DSKIP(8)  // will be initialized in the program, allocate 8 spaces

// And now, the program ... first, select single-precision
    CBON()

// Finish initialization of ArrayOfSevenPrimes, since we left out 11 above
// Use "add offset to the base" idiom (offset = index if we count from 1)
    SET(Rt, ArrayOfSevenPrimes) // base address of array
    SETLO(R1, 5)                // offset from that base, where we want 11
    ADD(Rt, Rt,R1)              // address where we want the 11
    SETLO(R1, 11)
    STORE(R1, 0,Rt)

// Now create ArrayOfSevenSquaredPrimes
// Use "keep relevant address in a register" idiom
    SET(R1, ArrayOfSevenPrimes) // R1 will be current place in 1st array
    SET(R2, ArrayOfSevenSquaredPrimes)  // R2 current place in 2nd array
    LOAD(R3, 0,R1)      // get size of ArrayOfSevenPrimes
    STORE(R3, 0,R2)     // set size of ArrayOfSevenSquaredPrimes

// while (--N_remaining >= 0)   // this is like a C-style while loop
  LABEL(SquareNextOne)
    DEC(R3, 1)          // R3 is now 1 less than the number remaining
  BLR(NoMoreSquares)    // If R3 is now less than 0, we are done.
// body of while loop
    INC(R1, 1)          // Move R1 (then R2) on to next memory cells
    INC(R2, 1)
    LOAD(R4, 0,R1)      // Get next element of ...Primes
    MUL(R4, R4,R4)      // Square it (note CB is on, for single-precision)
    STORE(R4, 0,R2)     // Set next element of ...SquaredPrimes
// end of while loop
  BRR(SquareNextOne)    // Go back and maybe do some more

// after while loop
  LABEL(NoMoreSquares)
  HALT()
```

Figure 6.2. Reading and Writing Length-Prefixed Arrays

6.3 Strings of Characters

High-level languages use a variety of mechanisms to represent strings of characters in memory, including sequences of non-null characters terminated by a null character (character 0), as in most C/C++ libraries; a length field followed by that many characters in the subsequent memory cells, as in Pascal and in Andrew Appel's "Tiger" language; and others. We could initialize either of these representations, or some other, with a collection of `INTEGER` data statements providing specific character values and/or size information.

For convenience, the assembler's `LP_STRING` data statement creates a sequence of character values preceded by a character count (as in Tiger, a string is an array of characters). Note that these strings still follow C-language conventions for escape characters. Figure 6.3 shows an example string processing program. The machine language for Figure 6.3's instructions would be `3160 e100 f100 e201 f2c0 4302 3280 4402 e53f f500 b045 0902 3180 3280 33c0 09f8 e233 f2c0 6102 0000`, and the data segment contains with the size (49, i.e., hexidecimal `0031`) and characters ('I', i.e., `0049`, 's', i.e., `0073`, and so on...) of `The_string`.

Hereafter, translation into machine language is left to any reader(s) interested in doing so.

```
// Count the number of times "?" appears in a String (single precision),
//  and save the result in the memory cell labelled N_questions.
// The example uses a constant string created with TIGER_STRING,
//  though the algorithm could be used for any string of this form.

  DLABEL(The_string)
    LP_STRING("Is this an example? With three questions? Really?")
  DLABEL(N_questions)
    INTEGER(0)

    CBON()
    SET(R1, 0)           // R1 will be the number of '?' found, initially 0
    SET(R2, The_string)  // R2 is now the address in memory of the string
    LOAD(R3, 0,R2)       // R3 = initial memory cell of a string (the length)
    INC(R2, 1)           // R2 = address of initial char. ('I', in this example)

  LABEL(top_of_loop)
    LOAD(R4, 0,R2)       // R4 is now the character itself
    SET(R5, 63)          // R5 is the character '?'
    SUB(R0, R4,R5)       // Set flags for the character (i.e., R4) - '?'
  BNZR(not_a_question)   // Skip the counting if R4-'?' is not zero
    INC(R1, 1)           // Count one question
  LABEL(not_a_question)
    INC(R2, 1)           // Move R2 to indicate the next character
    DEC(R3, 1)           // Decrease the count of how many remain
  BNZR(top_of_loop)      // If something other than 0 remain, keep going

    SET(R2, N_questions)
    STORE(R1, 0,R2)      // Store the result in memory
  HALT()
```

Figure 6.3. Counting the Number of Question Marks in a Tiger-style String

6.4 Records and Class-type Objects

A record or class-type object with n fields can be thought of as a collection of n values, similar to an array, except that we may not need to record the number of fields, and the fields may have different sizes in memory. Fortunately, all these things are typically known when accessing a field of an object.

For example, if a program represents of a `Point` class with a single-precision integer encoding, say, color, and then has two double-precision integers encoding x and y offsets, we could consistently use five consecutive memory cells for each object, with offset 0 for the color, offsets 1 and 2 for the x offset, and offsets 3 and 4 for the y offset. This fits well with the ability of HERA (and most RISC architectures) to combine a base address with a small constant offset, so that if we have the base address of a `Point` in R_2, and want to load its y field into R_6 and R_7, we could write `LOAD(R6, 3,R2) LOAD(R7, 4,R2)`.

Chapter 7
Assember Macros and HERA Functions

In a high-level language, a programmer can avoid repeatedly writing out a common group of steps by collecting them into a function (or procedure, or method, etc.) and then calling that function in several places. The language system can then reuse the steps of the function for each of the calls, via any of a number of possible mechanisms.

The mechanism that is usually discussed when function calls are taught, and shown when functions are explored in a debugger, involves transferring control to, and then back from, a single copy of the instructions for the function's body. Function call inlining is an alternative to this transfer-of-control mechanism, in which the body of the function is copied into each point at which the function is called.

In practice, many language systems employ some blend of these mechanisms, since the relative merits vary between functions. When inlining is performed before the program runs, e.g. by a classic compiler, it saves the run-time costs of branching to and from the function body. However, it can exponentially increase the size of the program, and simple implementactions cannot be applied to recursive functions or for dynamic dispacth (as used for overridden methods in object-oriented languages). Implementation of inlining may also require careful disambiguation of variable names that have different meanings in the called and calling function.

Assembly-language programmers can manually select a simple form of inlining by writing *macro*s rather than function definitions. Simple macros, such as those provided by the antiquated yet widely-known `#define` mechanism of the C preprocessor, perform textual substitution without disambiguating names. Note that the lack of automatic name disambiguation can lead to subtle bugs, e.g. the definition `#define sphereVolume(radius) (pow(radius, 3)*M_PI*4/3)` might seem like a foolproof way to find the radius of a sphere by using the C library definitions of pi (π) and pow (exponenentiation), until you (or a friend who is less familiar with the macro) use `sphereVolume` in code with a local varaiable named `M_PI` or `pow`.

The details of using macros and transfer-of-control for functions are discussed briefly in the upcoming sections of this chapter; for detail, consult a text on compiler design, e.g. [App98].

7.1 Macros in HERA-C and Hassem

HERA-C and Hassem borrow C's `#define` mechanism, along with its strengths and weaknesses. The problem of name confusion is particularly critical in assembly language code, where all code shares the same registers, register names, label names, and global data statements. Thus, macros are typically chosen over functions only for simple code that does *not* require additional registers, labels, etc. For example, a program that performs a lot of double-precision arithmetic might begin by defining `ADD_DP` and `ADD_SP` macros, each with three pairs of operands, as shown in Figure 7.1. Note that line breaks within macro definitions must be preceded by a backslash.

Since each use of a macro expands to the instructions given in the macro definitions, Figure 7.1 produces to the same sequence of 13 machine-language instructions as Figure 4.2, except that the latter places the literal value 15 into R_t instead of R_{10}, and does so after the first of the two parts of the addition of 1000000.

Functions are typically used when code requires additional registers, labels, or global data, or is in some way complex. Sometimes an exception is made when the run-time overhead of a call would pose a performance problem, in which case macros are used with great care. For a more extensive discussion of the use and perils of C's `#define` mechanism, consult references for the C programming langauge.

```
#define ADD_DP(d_hi,d_lo, a_hi,a_lo, b_hi,b_lo) \
  COFF() ADD(d_lo,a_lo,b_lo) ADD(d_hi,a_hi,b_hi)

#define SUB_DP(d_hi,d_lo, a_hi,a_lo, b_hi,b_lo) \
  CON()  SUB(d_lo,a_lo,b_lo) SUB(d_hi,a_hi,b_hi)


// Make [R1 R2] the (double-precision) sum of [R3 R4] + [R5 R6] + 1000000;
// finally, update [R7 R8] by subtracting the above total from it

CCBOFF()       // Enable use of carry flag for multiple-precision arithmetic
  // ...           (possibly other instructions that may set or clear the carry)

// start by finding [R3 R4] + [R5 R6]
ADD_DP(R1,R2, R3,R4, R5,R6)
// now add 1000000 to the above
SET(Rt, 16960) //  16960 is the lower 16 bits of 1000000, i.e., 1000000 %65536
SETLO(R10, 15) //  15    is the higer 16 bits of 1000000, i.e., 1000000//65536
ADD_DP(R1,R2, R1,R2, R10,Rt)

// next calculation: subtract the above from [R7 R8]
SUB_DP(R7,R8, R7,R8, R1,R2)
```

Figure 7.1. Figure 4.2 Rewritten to Use a Macro for Double-Precision Addition

7.2 Function Calls and Stack Frames

HERA, like most modern architectures, provides special instructions to assist with the transfer-of-control approach to function-call implementation. The HERA `CALL` instruction is used to branch to the start of a function; it differs from `BR` in that it updates the *frame pointer* and records the *return address*. The return address is the address of the instruction after the `CALL`, i.e., the instruction to which the program should branch when the function is done. The frame pointer (R_{14}, also known as FP) is used, along with the stack pointer (R_{15}, a.k.a. SP), to ensure that each function call has a distict *frame*, i.e., region of memory in which to record information specific to that call. The end of the called function uses the HERA `RETURN` instruction to transfer control back to the calling function by restoring the program counter (to the return address, which is also known as the *control link*) and frame pointer (to the frame pointer for the calling function, which is known as the *dynamic link* for the called function).

The exact nature and organization of stack frame information depends on the nature of the program being run and the algorithms used to implement high-level language techniques. The frame may contain some information that is evident even in a high-level language, such as values of local variables and parameters. The frame may also be used to store information that "links together" the different calls and frames, including the aforementioned control link and dynamic link, as well as the *static link*, which gives the frame pointer for the stack frame of the code that contains the code for function being executed. The static link is important when a function makes use of a variable from that surrounding code, for example, as the `two_a_plus_y` function of Figure 7.2 makes use of the variable `a` from the function `foo`, which contains the definitions of both `two_a_plus_y` and `a` (as `foo`'s first parameter). These links are usually put in standardized places in the frame, e.g. the return address in memory cell FP, the dynamic link in $FP + 1$, and the static link in $FP + 2$ (any arrangement can work as long as calling and called function are consistent).

Figure 7.3 shows what the stack frames for `foo` and `two_a_plus_y` might look like, if the instructions for the function `foo` run from instruction addresses 1000 to 1030, with the `CALL` to `two_a_plus_y` at instruction 1020, and if `foo` is executed with $FP = 100$ and $SP = 105$ (i.e., a five-element stack frame starting at 100). In this case, `two_a_plus_y` would have the five-element stack frame shown in the figure: the R.A. for `two_a_plus_y` will be 1021, since the `CALL` is instruction #1020; the D.L. will be 100, since `foo`'s stack starts there; the S.L. will also 100, since the source code for `two_a_plus_y` is contained within the code for `foo`; the value of

```
let foo(int a, int b): int =
    let two_a_plus_y(y : int): int =
        let two_a = a+a
        in  two_a+y
        end
    in
        two_a_plus_y(b-a+75) * a
    end
in
    printint(foo(2, 10))
end
```

Figure 7.2. Example Function Calls with Local Variables

`two_a_plus_y`'s parameter `y` will be 83,; and the value of the local variable `two_a` will be 4. The stack frame for `foo` would contain analogous information; since Figure 7.2 does not show the code that calls `foo`, the values of `foo`'s links are shown as "*???*".

This frame layout, together with the convention that FP will point to the beginning of the frame of the currently-executing function, lets each function access the data it needs. For example, `two_a_plus_y` could retrieve the value of its parameter `y` from its own stack frame into R_1 via the instruction `LOAD(R1, 3,FP)`; `two_a` could similarly be retrieved with offset 4. The static link provides access to `foo`'s variable `a`, via e.g., `LOAD(R2, 2,FP) LOAD(R2, 3,R2)`. Although the static and dynamic links are identical for the code of Figure 7.2, this is not always the case, e.g. if `foo` had called another function that had, in turn, called `two_a_plus_y`.

There are a number of subtle details involved in implementing each approach to organizing stack frames, so the full detail will be presented after discussion of some simple cases.

7.3 Simple Cases: Calls to Trivial or Library Functions

For some simple kinds of function calls, we may not have to build and use a complete stack frame. For example, if a function is to be used to perform specific updates to specific hardware registers, the purpose of `CALL` and `RETURN` is simply to transfer control into the function and back. When we call a pre-written function from a library, we must initialize the stack frame in a way that is consistent with the function's body, but we need not write the body of the function or be familiar with the elements of the frame that will be updated within the function.

The remainder of this section will provide examples of basic usage of HERA's `CALL` and `RETURN` instructions for these cases; the subsequent sections provide more detail about the techniques used to create the function bodies. Throughout this chapter, we will follow conventions used for/by the HERA-C system and the Hassem assembler. Specifically, in each `CALL` and `RETURN`, we will use R_{12} for the new value of the frame pointer, and R_{13} as the branch address.

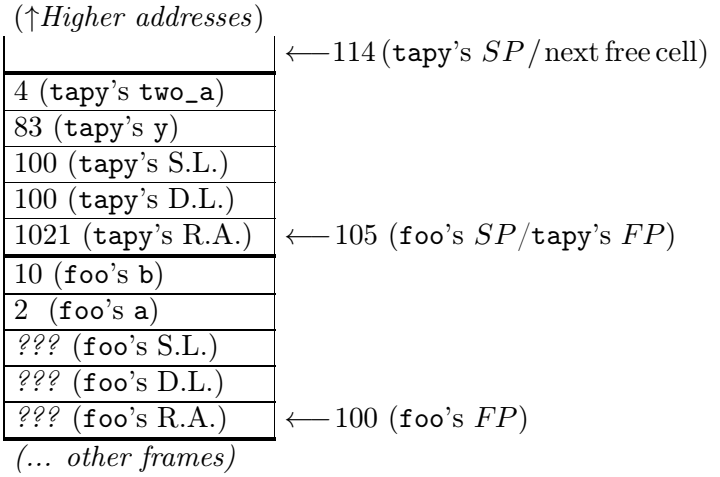


Figure 7.3. Frames for a Sample Function Call Stack

To emphasize these conventions, HERA-C and Hassem provide the names `PC_ret` for the branch address/return address (R_{13}) and `FP_alt` for the alterate frame pointer (R_{12}). Proper idiomatic usage is thus always `CALL(FP_alt, label)` and `RETURN(FP_alt, PC_ret)`.

7.3.1 A Trivial Function and a Call Thereof

Figure 7.4 illustrates the use of HERA's `CALL` and `RETURN` for a trivial function that adds $2R_1 + R_2$ to the existing value of R_3. Since the purpose of this function is simply to update some registers, we do not need a stack frame for parameters or local variables, and can ignore the role of the frame pointer and `FP_alt` for now.

Recall that the assembler automatically converts labels into address numbers; since the label `update_r3` precedes Instruction 12, the assembler turns each `CALL(FP_alt, updateR3)` into the three-instruction sequence `SETLO(`R_{13}`, 12) SETHI(`R_{13}`, 0) CALL(`R_{12}`, `R_{13}`)`. Since R_{13} and the upcoming the value for the PC are exchanged by `CALL(`R_{12}`, `R_{13}`)`, the call at Instruction 5 will save 6 into R_{13} while branching to Instruction 12, and the call at Instruction 10 will save 11 in R_{13} while branching to Instruction 12. In either case, the HERA processor will then execute Instructions 11-13, followed by the `RETURN` at instruction address 14. Since `RETURN(`R_{12}`, `R_{13}`)` does the same $R_{13} \leftrightarrows$ upcoming-PC switch as `CALL(`R_{12}`, `R_{13}`)`, but R_{13} now has the value 6

```
// This program demonstrates function calls, using one simple
//    function to repeatedly add 2*R1+R2 into existing value of R3,
// First with R1=100,R2=50, then with R1=10,R2=3.

    CBON()              // Instruction 0

    SETLO(r1, 100)      // Instruction 1
    SETLO(r2,  50)      // Instruction 2
    // Auto-generated instructions 3&4 SETLO/SETHI(R13, updateR3)
    CALL(FP_alt, updateR3)  // Instruction 5: CALL(R12, R13)

    SETLO(r1, 10)       // Instruction 6
    SETLO(r2,  3)       // Instruction 7
    // Auto-generated instructions 8&9 SETLO/SETHI(R13, updateR3)
    CALL(FP_alt, updateR3)  // Instruction 10: CALL(R12, R13)

    HALT()              // Instruction 11

// The body of updateR3:
//   compute 2*r1 + r2, leaving the result in r3
LABEL(updateR3)
    ADD(r1, r1,r1)      // 2*r1         (Instruction 12)
    ADD(r1, r1,r2)      // 2*r1+r2      (Instruction 13)
    ADD(r3, r3,r1)      // r3+=2*r1+r2  (Instruction 14)
    RETURN(FP_alt, PC_ret)  // go back  (Instruction 15)
```

Figure 7.4. Simple Function and Call, Showing HERA `CALL` and `RETURN` Instructions.

(during the first call) or 11 (during the second call), the `RETURN` will branch back to *either* Instruction 6 or 11, arriving at the proper place after each call.

Note that we could also achieve the effect of our `updateR3` with a macro; the most direct equivalent would be `#define updateR3() ADD(R1,R1,R1) ADD(R1,R1,R2) ADD(R3,R3,R1)`. For that definition, we would write `updateR3()` instead of the `CALL` instructions of Figure 7.4. If we wanted to provide more flexiblity, we could, for example, let the program add $2R_1 + R_2$ to any register, via `#define updateAny(Any) ADD(R1,R1,R1) ADD(R1,R1,R2) ADD(Any,Any,R1)`, and write `updateAny(R3)` instead of the call. To create an a more flexible function, we will need to use techniques from the upcoming sections.

7.3.2 Calling Library Functions that Expect Parameters in Registers

One convention that is often adopted for functions is to place the parameters values in registers $R_1...R_n$ (for an n-parameter function, assuming $n \leqslant 10$), and place the returned value (if any) in R_1 (or, m values can be returned in $R_1...R_m$, assuming $m \leqslant 10$). In this case, the functions may or may not also identify some registers that will remain unchanged. While the bodies of such functions are more complex than that of `updateR3`, calls to them are just as easy, as long as we respect the conventions when we choose which registers will be used for which values in our code and ensure that `FP_alt` is initialized to the value of `SP` so that the function starts with a valid empty stack frame.

Figure 7.5 illustrates the use of this convention for a program that calls library functions (not shown) for computing integer quotients and for printing integers and strings. The expectations about register usage are documented in the comments, and must be consistent with the actual function definitions (in particular, that R_1 and R_2 are used for parameters 1 and 2, that the return value is placed in R_1, and that R_2 and R_4 are not changed by any functions we call). Much of the challenge in using this idiom lies in ensuring that the necessary value is in the necessary register at each step of the computation.

7.3.3 Calling Library Functions that Expect Parameters in the Stack

When there are not enough registers to hold all the parameters and the other local variables, we must place some values into memory. One approach to doing so involves creating a stack frame for the function to be called, and pre-filling it with the values of the parameters. In this approach, as with the previous, the calling function must be consistent with the called function. One additional subtletly is that the called function may require that some initial elements of the stack frame are skipped before the first parameter, to provide storage for the information that is needed to link together the function call sequence, i.e., to get back from the called function to the data or execution of the calling function.

Figure 7.6 shows a variant of Figure 7.5 that is written for functions that use this approach, reserving the first three spaces of each frame for the "linking together" information, using frame elements 3 and above for parameters and the return value, and preserving register values. This idiom involves some additional lines of code, in particular creating a stack frame of size $n + 3$ for a function with n parameters.

```
// Use library functions "div", "print", and "printint" to
//   print a quotient calculation, e.g. "210//5 = 42"
#include <Tiger-stdlib-reg-data.hera>

// First, global variables, e.g. INT_DIVIDE_AS_IN_PYTHON="//"
DLABEL(INT_DIVIDE_AS_IN_PYTHON)     LP_STRING("//")
DLABEL(EQUALS_SIGN_WITH_SPACES)     LP_STRING(" = ")

// initialize CB, R1, and R2; any single-precision values are fine for R1,R2
    CBON() // library functions may assume CB is set, in this case
    SET(R1, 210)
    SET(R2, 5)

// Now, the algorithm:
//  printint(R1)
//  print(INT_DIVIDE_AS_IN_PYTHON)
/// printint(R2)
//  print(EQUALS_SIGN_WITH_SPACES)
//  printint(div(R1, R2))

    MOVE(FP_alt, SP)            // put called function's frame above this one
    CALL(FP_alt, printint)      // printint prints single-precision integer in R1

// Below, we assume R4 and R5 are not changed in library functions we call
    MOVE(R4, R1)                // Keep old R1 in R4 while we use R1 for "//"
    MOVE(R5, R2)                //   likewise, keep R2 around...
    SET(R1, INT_DIVIDE_AS_IN_PYTHON)
    CALL(FP_alt, print)         // print prints an LP_STRING given by R1

    MOVE(R1, R5)                // now print original R2, which we saved in R5
    CALL(FP_alt, printint)

    SET(R1, EQUALS_SIGN_WITH_SPACES)
    CALL(FP_alt, print)         // print " = "

    MOVE(R1, R4)                // now, we'll need that original value of R1
    MOVE(R2, R5)                //       and the original value of R2
    CALL(FP_alt, div)           // div finds R1//R2, changes R1 to that
    CALL(FP_alt, printint)      // conveniently, R1//R2 is where printint wants it

    HALT()

#include <Tiger-stdlib-reg.hera>
```

Figure 7.5. Calls to Library Functions that Use $R_1 ... R_2$ For Parameters and Do Not Change R_4 or R_5

```
// Use library functions "div", "print", and "printint" to
//   print a quotient calculation, e.g. "210//5 = 42"
#include <Tiger-stdlib-stack-data.hera>

DLABEL(INT_DIVIDE_AS_IN_PYTHON)     LP_STRING("//")
DLABEL(EQUALS_SIGN_WITH_SPACES)     LP_STRING(" = ")

    CBON() // library functions may assume CB is set, in this case
    SET(R1, 210)        // any single-precision values are fine for R1,R2
    SET(R2, 5)

// Now, the algorithm:
    MOVE(FP_alt, SP)            // printint's frame in free space above this frame
    INC(SP, 4)                  //  printint frame starts at size 4
    STORE(R1, 3,FP_alt)         //  printint's first param. in FP+3
    CALL(FP_alt, printint)      // printint prints single-precision int. in FP+3
    DEC(SP, 4)                  //  put SP back where it was

    MOVE(FP_alt, SP)            // Could omit this here; FP_alt is already = SP
    INC(SP, 4)                  //  could omit this if we'd left out DEC above
    SET(Rt, INT_DIVIDE_AS_IN_PYTHON)
    STORE(Rt, 3,FP_alt)         // print's first param. in FP+3
    CALL(FP_alt, print)         // print prints an LP_STRING given by FP+3

    STORE(R2, 3,FP_alt)         // print R2 this time; FP_alt and SP as above
    CALL(FP_alt, printint)

    SET(Rt, EQUALS_SIGN_WITH_SPACES)
    STORE(Rt, 3,FP_alt)         // print " = ", FP_alt and SP as above
    CALL(FP_alt, print)

    INC(SP, 1)                  // starting frame size is 5 (3, plus 2 param.)
    STORE(R1, 3,FP_alt)         //   first parameter at FP_alt+3
    STORE(R2, 4,FP_alt)         //   first parameter at FP_alt+4
    CALL(FP_alt, div)           // div puts mem(FP+3)//mem(FP+4) in mem(FP+3)
    DEC(SP, 1)                  // starting frame size is 4 for printint again
    CALL(FP_alt, printint)      // conveniently, R1//R2 is where printint wants it
    DEC(SP, 4)                  // put things back where they started

    HALT()

#include <Tiger-stdlib-stack.hera>
```

Figure 7.6. Calls to Library Functions that use Frame Locations 3 and 4 for Parameters

7.4 Parameters in Registers, "Caller-Save" Registers

The simplest approach to writing functions that expect parameters in registers is to make no guarantees at all about what becomes of the registers other than the one used for the return value. This "functions may change register values" approach forces the caller to save, before each `CALL` instruction, any register values that may be needed after the call. It is thus referred to as a "caller-save" convention. Note that the called function will need to preserve the return address and dynamic link (otherwise, it could not return correctly); by convention, caller-save functions in HERA preserve $R_{12}...R_{15}$. When the function does not, itself, do any function calls, it can simply rely on the fact that these registers will remain the same because they have not been changed. Figure 7.7 shows the definition of a function `two_x_plus_y`, which returns twice its first parameter plus its second using this convention; `two_x_plus_y` is similar to `updateR3` of Figure 7.4, in that it computes $2R_1 + R_2$, but it returns this value instead of adding it to R_3.

Note that the burden of saving registers can complicate the code for making a function call; instead of using the simple `CALL` and `RETURN` statements of Figure 7.4 and 7.5, we'll need to create space in the stack for any variables we'll need to preserve, store the needed values there before the call, and load them back into registers afterwards. Figure 7.8 shows this style of call for a HERA program equivalent to `x=5; printint(foo(x+5,2)-x)`; note that the value of `x`, which was originally in R_1, is saved to memory cell $FP + 0$ before the call to `foo`, and then loaded back into R_2 for the subtraction step. No such save/reload sequence is needed for the call to `printint`, however, since `x` is not used after that point.

If a function makes many calls in exactly this way, it can end up moving the stack pointer up and down, and saving and re-loading values more than is necessary. We avoid this wasted work by allocating enough space when the function starts, and saving and loading only as needed.

Putting this all together, we adopt a convention in which we call a function with these steps:

- Save (into the stack frame) any registers we may need after the call and whose values have not yet been saved
- Put the parameter values into registers R_1, R_2, ...
- Copy the current stack pointer into R_{12}, i.e., via `MOVE(`R_{12}`, SP)`, unless a prior call did so
- Actually call the function, via `CALL(`R_{12}, *label-for-start-of-function*`)`
- After the call, the returned value can be retrieved from R_1
- Load (from the stack frame) any of the saved registers that we need before the next call.

```
// Translation of a high-level-language function two_x_plus_y:
//     int two_x_plus_y(x : int, y : int) : int = x+x+y
// assuming parameters/return in registers, no static link, caller-save of reg.,
// (as before single precision, assuming CB set)

LABEL(two_x_plus_y)
    // Don't bother saving Rt since it won't be changed
    //  (no "NOT", assembler-generated register mode branches, etc)
    ADD(r1, r1,r1)       // result = 2*x
    ADD(r1, r1,r2)       // result = 2*x+y
    RETURN(FP_alt, PC_ret)
```

Figure 7.7. Function `two_x_plus_y`

To define a function to be called with these conventions, we use these steps:

- If necessary, increment SP enough to make space for saving registers during the function (for functions of any significant length, it may be useful to document, with a comment, what will go at each memory cell in the frame, i.e., what's at $FP+0$, at $FP+1$, etc.).
- If necessary, save the return address (from R_{13}) and dynamic link (from R_{12}) in the frame
- Give the function body (in which parameters are assumed to be in R_1, R_2, ...)
- Put the function's return value into R_1
- If necessary, load the return address (into R_{13}) and dynamic link (into R_{12}) from the frame
- If necessary, decrement SP to restore its value prior to the function
- Return from the function, via `RETURN(`R_{12}, R_{13}`)`.

Figure 7.9 shows the use of these conventions to define a function `foo`, which calls `two_x_plus_y`. Figure 7.10 shows the memory layout of the stack frames for this program, with the values that are there during the execution of `two_x_plus_y`.

This set of conventions is appealing for simple programs like those shown here, but must be extended carefully for programs in which some functions have more than 11 parameters, or for languages in which one function definition can be nested within another and the inner function can access the variables of the outer (as in Figure 7.2). Such functions can be handled using the techniques of the next section, or a carefully crafted hybrid of these two approaches.

```
// Example program x=5; printint(foo(x+5, 2)-x)

CBON()

// Assume we start with a size-zero stack frame, i.e., SP=FP
SET(r1, 5)          // use r1 for x, set x=5


// call foo(x+5, 2) ... start by saving x, which we'll need later
INC(SP, 1)          // SP is now 1 above FP; we have a 1-cell frame
STORE(r1, 0,FP)     // Save x at offset 0 in our frame

INC(r1, 5)          // r1 (first parameter) is now X+5
SET(r2, 2)          // r2 (second parameter) is 2
MOVE(FP_alt, SP)    // New frame (for foo) should be current SP
CALL(FP_alt, foo)   // call foo, use FP_alt to indicate new FP

LOAD(r2, 0,FP)      // get x back into r2; note r1 is result of foo
DEC(SP, 1)          // clean up stack, since we're done with it
SUB(r1, r1,r2)      // find result of foo - x, i.e. r1-r2

CALL(FP_alt, printint)  // print the result
HALT()
```

Figure 7.8. Calls to `foo` and `printint`, with Parameters in Registers

```
// Translation of foo, assuming parameters in registers etc.
//     int foo(a : int, b : int) : int = two_x_plus_y(a+b, b-a+75) * a

LABEL(foo)   // Assumes carry-block is set
// FIRST, make space to save R.A., old FP, and "a", then save them
    INC(SP, 3)
    STORE(PC_ret, 0,FP) // Save R.A.
    STORE(FP_alt, 1,FP) // Save old F.P. ("control link")
    STORE(r1, 2,FP)     // Save "a" in current frame (only re-save if changed)

// copy "a" and "b" out of r1 and r2 so that we can use r1 and r2 in next call:
    MOVE(r10, r1)
    MOVE(r11, r2)
// set up parameters for call to two_x_plus_y
    ADD(r1, r10,r11)    // r1 = a+b
    SUB(r2, r11,r10)    // r2 = b-a
    SETLO(r9, 75)
    ADD(r2, r2,r9)      // r2 = b-a+75
// actually make the call, with two_x_plus_y's parameters in r1 and r2
    MOVE(FP_alt, SP)    // initial stack frame for called function has size 0
    CALL(FP_alt,two_x_plus_y)
// Now restore "a" and multiply result of call (now in r1) by it:
    LOAD(r2, 2,FP)
    MUL(r1, r1,r2)

// Finally, restore R.A. and old F.P., do the return (result is already in r1)
    LOAD(PC_ret, 0,FP)
    LOAD(FP_alt, 1,FP)
    DEC(SP, 3)
    RETURN(FP_alt, PC_ret)
```

Figure 7.9. Calling `two_x_plus_y` with Parameters in Registers

(↑*Higher addresses*)

	⟵4 (`two_x_plus_y`'s FP=`foo`'s SP)	size zero frame for `two_x_plus_y`
10		`foo`'s variable `a`
0		`foo`'s C.L. (saved FP of main)
??? (P.C. in main)	⟵1 (`foo`'s FP=main's SP=1)	`foo`'s R.A. (P.C. in main)
5	⟵0 (main's FP=0)	value of X

Figure 7.10. Active Stack Frames During Execution of `two_x_plus_y` From Figure 7.8's Main Program

7.5 Parameters on the Stack, "Callee-Save" Registers

The previous section's use of registers for function parameters does not work well for functions with very large numbers of parameters, or when functions refer to variables from outer scopes (as with the variable `a` of `two_a_plus_y` in Figure 7.2). The caller-save approach to registers simplifies function bodies, at the expense of complexity for function calls. Either of these conventions can be varied; this section illustrates the use of the stack frame for parameters and a callee-save approach to preserving register values, and follows the convention that the first three elements of each frame are reserved for the links, as in Figures 7.3 and Figure 7.6. The callee-save approach moves the burden of saving register values from the calling function to the called function, which need only save the registers it will change (except for R_t, by convention). Other combinations of parameter-passing mechanism and register preservation are possible, as are hybrids of these approaches.

Figure 7.11 shows a version of our `two_x_plus_y` example with the "parameters on the stack, callee-save of registers" conventions. Since it has two parameters, it should be called with a stack frame of size five that contains the values of `x` and `y` (in memory cells $FP + 3$ and $FP + 4$, respectively). The two `ADD` steps in the middle of Figure 7.11 are analogous to the `ADD`s of Figure 7.7. Since the parameters and return value are passed in the stack, the additions are preceded by

```
// Translation of a high-level-language function two_x_plus_y:
//     int two_x_plus_y(x : int, y : int) : int = x+x+y

LABEL(two_x_plus_y)
// FIRST, make space to save r1 and r2 and then save PC_ret, FP_alt, and them
    INC(SP, 2)
    STORE(PC_ret, 0,FP) // Return address, i.e. PC+1 from before CALL
    STORE(FP_alt, 1,FP) // Control Link, i.e. FP from before the CALL
    STORE(r1, 5,FP)     // skip FP+3 and FP+4, where a and b will be...
    STORE(r2, 6,FP)     //   ... and save r1 and r2 in FP+5/FP+6

// Load "x" and "y" from stack frame (calling func. put them there)
    LOAD(r1, 3,FP)      // r1 = x
    LOAD(r2, 4,FP)      // r2 = y

// Compute the result
    ADD(r1, r1,r1)      // r1 = x+x
    ADD(r1, r1,r2)      // r1 = x+x+y

// Store the result where the caller of two_x_plus_y will find it
    STORE(r1, 3,FP)

// FINALLY, restore registers (including PC_ret and FP_alt) and return
    LOAD(r2, 6,FP)      // Restore r2
    LOAD(r1, 5,FP)      // Restore r1
    LOAD(PC_ret, 0,FP)  // Restore PC_ret to provide R.A. for return
    LOAD(FP_alt, 1,FP)  // Restore FP_alt (D.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(FP_alt, PC_ret)
```

Figure 7.11. A Function that Retrieves Parameters from the Stack and Preserves Register Values

`LOAD` instructions to retrieve `x` and `y` from the stack frame, and followed by a `STORE` to put the return value into the stack frame (where the first parameter had been). To respect the callee-save approach to register values, `two_x_plus_y` stores the values of R_1 and R_2 in the frame at the start of the function and loads them back at the end; the `INC(SP, 2)` and `DEC(SP, 2)` lines increase the stack size at the start and restore it at the end. This function also saves and restores `FP_alt` and `PC_ret`, though this is not necessary in this particular function since it does not change these registers (some programmers choose to include these sometimes-unnecessary steps in every function as a form of "defensive programming", to avoid having to debug code in which they had been inappropriately omitted).

Figure 7.12 provides examples of calling functions with these conventions; like Figure 7.8, it is a HERA equivalent of `x=5; printint(foo(x+5,2)-x)`. Unlike Figure 7.8, it can rely on R_1's value is unchanged during `foo`, thanks to the callee-save convention. Note the steps involved in establishing the proper size stack frame, storing parameter values, and retrieving the return value.

```
// Example main program:   x=5; printint(foo(x+5, 2)-x)

CBON()

// Assume we start with a size-zero stack frame, i.e., SP=FP
SET(r1, 5)              // use r1 for x, set x=5

// call foo(x+5, 2)
MOVE(FP_alt, SP)        // New frame (for foo) should be current SP
INC(SP, 5)              // foo has 2 pameteters, start with stack size 5
MOVE(R2, R1)            // R2 is x, we'll increment it to x+5
INC(R2, 5)
STORE(R2, 3,FP_alt)     // parameter 1 = x+5
SET(r2, 2)
STORE(r2, 4,FP_alt)     // parameter 2 = 2
CALL(FP_alt, foo)
LOAD(R2, 3,FP_alt)      // retrieve returned value into R2
DEC(SP, 5)              // free up memory from foo's frame

// call printint(foo's result - x)
SUB(R2, R2,R1)          // R2-R1 is foo's return - x, i.e., foo(x+5,2)-x
INC(SP, 4)              // printint has 1 parameter, start with stack size 4
STORE(R2, 3,FP_alt)     // put printint's parameter in place
CALL(FP_alt, printint)  // print the result
DEC(SP, 4)

HALT()
```

Figure 7.12. Main Program to Call a Function `foo`, Passing Parameters on the Stack

To call a function that uses this convention, we:

- Set `FP_alt`$\leftarrow SP$ and increment SP to allocate initial stack frame (size 3 + #parameters)
- Put the parameters on the stack above 3 spaces for links (i.e., starting at `FP_alt` +3)
- Set up the static link, if one is needed (see discussion later in this section)
- Issue the `CALL` instruction
- After the call, the return value can be retrieved from `FP_alt` +3, and SP decremented.

To define a function to be called with these conventions, we use these steps, as needed:

- Increment SP to make space for local storage
- Save registers, including `PC_ret` (return address) and `FP_alt` (dynamic link)
- Give the function body (in which parameters come from the stack frame, e.g. $FP + 3$)
- Store the return value at $FP + 3$
- Restore saved registers, including `FP_alt` and `PC_ret`, and decrement SP
- `RETURN` from the function

Thus, when one function calls another using these conventions, the caller creates the initial frame for the called function within the top of its own frame, as illustrated by the shaded region of Figure 7.13 (for a call to a k-parameter function from an n-parameter function that uses m memory cells for storage within the function, e.g. for local variables or saved registers). The shaded region becomes the initial frame for the called function as the `CALL` instruction exchanges FP and `FP_alt` (the frame for the called function may then grow as the function runs).

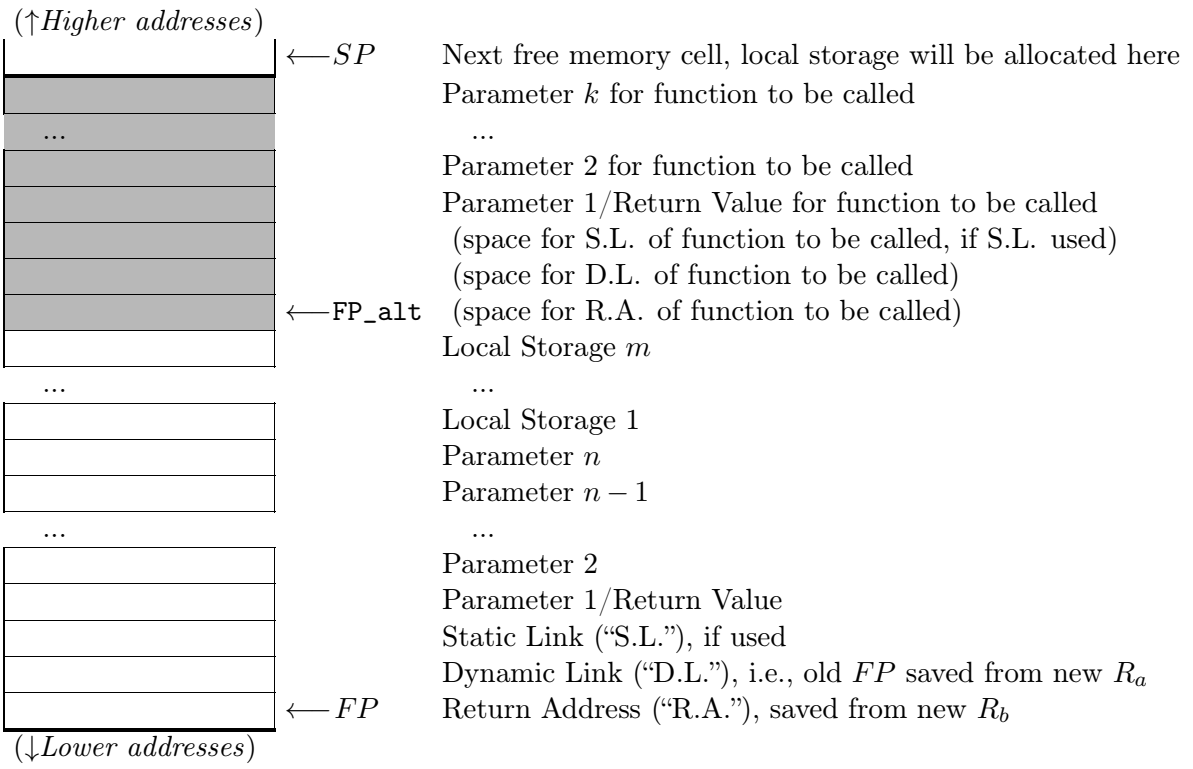


Figure 7.13. Typical Stack Frame with Parameters and Return Value on Stack, Just Before a `CALL`

```
// Translation of foo (single precision, assuming CB set)
//     int foo(a : int, b : int) : int = two_x_plus_y(a+b, b-a+75) * a

LABEL(foo)
// FIRST, make space to save r1 and r2 and then save PC_ret, FP_alt, and them
    INC(SP, 2)
    STORE(PC_ret, 0,FP) // Return address
    STORE(FP_alt, 1,FP) // Control Link
    STORE(r1, 5,FP)     // skip FP+3 and FP+4, where a and b will be
    STORE(r2, 6,FP)

// THEN the body of foo: compute two_x_plus_y(a+b, b-a+75) * a
//    For the call, create space for 3 links and 2 parameters,
//    remembering old SP in FP_alt:
    MOVE(FP_alt, SP)
    INC(SP, 5)
//    and set up parameters a+b and b-a+75
    LOAD(r1, 3,FP)      // R1 = a (from memory cell FP+3)
    LOAD(r2, 4,FP)      // R2 = b
    ADD(Rt, r1,r2)      // Rt = a+b
    STORE(Rt, 3,FP_alt) //  1st parameter at FP_alt+3, i.e. CALLED FUNC'S FP+3
    SUB(r2, r2,r1)      // R2 = b-a
    SETLO(Rt, 75)
    ADD(r2, r2,Rt)      // R2 = b-a+75
    STORE(r2, 4,FP_alt) //  establish 2nd parameter at FP_alt+4

//    and do the call
    CALL(FP_alt, two_x_plus_y)

// AFTER the call, retrieve result and multiply by "a"
    LOAD(r2, 3,FP_alt)  // R2 = result retured FROM CALLED FUNC'S FRAME
//    now that we have the returned value, we can shrink the stack back down:
    DEC(SP, 5)          // could use MOVE(SP, FP_alt)
// NOTE that r1 is still "a" from before the call:
    MUL(r1, r2,r1)

// Save the result where the caller of 'foo' will find it
    STORE(r1, 3,FP)     // Put return value over 1st parameter

// FINALLY, restore registers (including PC_ret and FP_alt) and return
    LOAD(r2, 6,FP)      // Restore r2
    LOAD(r1, 5,FP)      // Restore r1
    LOAD(PC_ret, 0,FP)  // Restore PC_ret to provide return address for return
    LOAD(FP_alt, 1,FP)  // Restore FP_alt (D.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(FP_alt, PC_ret)
```

Figure 7.14. Calling A Function, with Parameters on Stack

Figure 7.14 shows the definition of a function `foo` that calls `two_x_plus_y`, as illustrated by pseudo-code above the function. Since `foo` is a function that can be called, it includes the "to define a function" steps above, as did Figure 7.11 (but, in this case, the save and restore of `FP_alt` and `PC_ret` are necessary for the program to work). Since `foo` makes a call, it also includes the "to call a function" steps, as did Figure 7.12. We could optimize Figure 7.14 by combining the two increments of SP, and likewise combining the two decrements. This would slightly shorten the function, possibly at the cost of some clarity.

The `two_x_plus_y` example above does not illustrate the actual use of a static link. Figures 7.15 and 7.16 show a HERA version of the functions `foo` and `two_a_plus_y` of Figure 7.2, in which `two_a_plus_y` uses `foo`'s local variable `a`. The algorithms for setting up and using static links are beyond the scope of this document, but a careful examinition of Figures 7.15 and 7.16 should illustrate how the static link is used in this example: the static link for `two_a_plus_y` is established as the base address of `foo`'s frame, right before `foo` calls `two_a_plus_y`; this lets `two_a_plus_y` find the value of `a` by retrieving `foo`'s frame pointer and then loading the variable at offset 3.

Figure 7.17 shows a memory layout diagram of the stack during the execution of this program, at the instant just before the `DEC(SP,2)` at the end of `two_a_plus_y`, assuming `foo` is called by the code of Figure 7.12 and that this code is executed when the HERA microprocessor starts up, with all registers (including FP and SP) being 0, and thus with a zero-size stack frame for the main program. (Memory cells with contents marked "*" are addresses of instructions in the program—these can't be determined without knowing how the code from the various figures is laid out in the address space of the instruction memory).

```
// Translation of two_a_plus_y, using lexically scoped variable "a"
LABEL(two_a_plus_y)
    INC(SP, 2)
    STORE(PC_ret, 0,FP) // R.A.
    STORE(FP_alt, 1,FP) // D.L.
    STORE(r1, 4,FP)     // Save r1 and r2 at FP+4 and FP+5,
    STORE(r2, 5,FP)     //  because there is only one parameter (y)

// Load "a" from offset 3 of statically scoped frame
    LOAD(r1, 2,FP)      // get two_a_plus_y's static link, i.e., foo's FP
    LOAD(r1, 3,r1)      // r1 now is "a"

// Load "y" from two_a_plus_y's stack frame (as in two_x_plus_y example)
    LOAD(r2, 3,FP)      // r2 = y, from current frame

    ADD(r1, r1,r1)      // Compute the result: r1 = a+a
    ADD(r1, r1,r2)      // r1 = a+a+y

    STORE(r1, 3,FP)     // Store the result
    LOAD(r2, 5,FP)      // Restore r2
    LOAD(r1, 4,FP)      // Restore r1
    LOAD(PC_ret, 0,FP)  // Restore PC_ret (R.A.)
    LOAD(FP_alt, 1,FP)  // Restore FP_alt (D.L.)
    DEC(SP, 2)
    RETURN(FP_alt, PC_ret)
```

Figure 7.15. Function `two_a_plus_y`, Illustrating the Use of the Static Link (see also Figure 7.16).

```
// Translation of main function foo, setting up to use lexical scoping
//     foo(a: int, b: int): int =
//          let two_a_plus_y(y : int): int = a+a+y
//          in  two_a_plus_y(b-a+75) * a
// (once again, assuming single precision, and carry-block is set)

LABEL(foo)
// Standard preamble
    INC(SP, 2)
    STORE(PC_ret, 0,FP)
    STORE(FP_alt, 1,FP)
    STORE(r1, 5,FP)
    STORE(r2, 6,FP)

// Allocate space for new frame, define ONE parameter "y"
    MOVE(FP_alt, SP)
    INC(SP, 4)

    LOAD(r1, 3,FP)      // R1 = a
    LOAD(r2, 4,FP)      // R2 = b
    SUB(r2, r2,r1)      // R2 = b-a
    SETLO(Rt, 75)
    ADD(r2, r2,Rt)      // R1 = b-a+75
    STORE(r2, 3,FP_alt)

// Build the static link for two_a_plus_y (points to foo's frame),
    STORE(FP, 2,FP_alt) // CREATE STATIC LINK FOR two_a_plus_y
    CALL(FP_alt,two_a_plus_y)

    LOAD(r2, 3,FP_alt)  // Retrieve the result
    DEC(SP, 4)

    MUL(r1, r1,r2)      // Multiply by "a"
    STORE(r1, 3,FP)     // Save result as return value

// Standard postable
    LOAD(r2, 6,FP)
    LOAD(r1, 5,FP)
    LOAD(PC_ret, 0,FP)
    LOAD(FP_alt, 1,FP)
    DEC(SP, 2)
    RETURN(FP_alt, PC_ret)
```

Figure 7.16. Function Calls with Parameters on Stack and an Escaping Local Variable

(↑*Higher addresses*)		
	⟵13 (`two_x_plus_y`'s SP)	Next free memory cell
67		`two_a_plus_y`'s saved R_2
10	⟵11 (`two_x_plus_y`'s initial SP)	`two_a_plus_y`'s saved R_1
91 (was 12)		`two_a_plus_y`'s result (was `x`)
0		`two_a_plus_y`'s S.L. (`foo`'s FP)
0		`two_a_plus_y`'s D.L. (`foo`'s FP)
*	⟵7 (`two_x_plus_y`'s FP)	`two_a_plus_y`'s R.A. (into `foo`)
0		`foo`'s saved R_2
0	⟵5 (`foo`'s initial SP)	`foo`'s saved R_1
10		`foo`'s parameter `b`
2		`foo`'s parameter `a`
		`foo`'s S.L. (undefined)
0		`foo`'s D.L. (saved FP of main)
*	⟵0 (`foo`'s FP=0)	`foo`'s R.A. (into main)

Figure 7.17. Stack Layout for Figures 7.14, 7.11, and 7.12 Just Before `DEC(SP, 2)` in `two_x_plus_y`.

7.6 Hybrid Conventions

Conventions for placement of parameters, saving of registers, choice of which registers to use, etc., can be combined in a variety of other ways. These choices must be made with respect to the programming language being used and degree of analysis/optimization performed by the compiler or programmer, as well as the target architecture.

The example code of Figure 7.5 uses the convention that the first three parameters are passed in R_1, R_2, and R_3; that $R_4 ... R_7$ must be preserved after a function's execution (i.e., they must be saved and restored by the called function if they are changed), but $R_8 ... R_{11}$ may be changed; and that $R_{12} ... R_{15}$ will, like $R_4 .... R_7$, be preserved after function's execution. This makes $R_4 .... R_7$ suitable for long-term storage of variables, and $R_8 .... R_{11}$ suitable for short-lived temporaries.

For more information about ways of combining the these conventions, refer to a compiler implementation textbook. Enthusiastic readers are invited to program the examples of this chapter using the hybrid convention above.

Bibliography

[App98] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[HP07] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[Man88] M. Morris Mano. *Computer Engineering: Hardware Design*. Prentice Hall, 1988.

[Uni12] Unicode, Inc. The Unicode standard 6.1. http://www.unicode.org/charts/PDF/U0000.pdf, 1991-2012.

[Won06] David Wonnacott. Unifying the undergraduate applied CS curriculum around a simplified microprocessor architecture. In *Proceedings of the 22nd Annual Consortium for Computing Sciences in Colleges Eastern Conference (CCSC-E 06)*, October 2006.