The Haverford Educational RISC Architecture

Table of contents

Table of contents 2

List of figures 3

1 Purpose and History 3

2 Architecture Overview 4

3 Instruction Set 4

3.1 Arithmetic, Shift, and Logical Instructions ($b_{15} = 1$, $b_{15:12} = 0011$) 4
3.1.1 SETLO and SETHI ($b_{15:13} = 111$) 4
3.1.2 Three-address operations ($b_{15:13} = 110$, 101, and 100) 5
3.1.3 Shifts, increments, and flag operations ($b_{15:12} = 0011$) 5
Shifts 5
Set/clear flags 5
Save/restore flags 6
Increments 6
3.2 Memory Instructions ($b_{15:14} = 01$) 7
3.3 Control-Flow and Other Instructions ($b_{15:14} = 00$) 7
3.3.1 Branches, including jumps ($b_{15:13} = 000$) 8
Special branches 8
3.3.2 Function calls ($b_{15:12} = 0010$) and returns (0001 0001 0001 0001) 9

4 Assembly Language Conventions and Pseudo-Operations 10

5 Idioms 10

5.1 Single-Precision Arithmetic 10
5.2 Double-Precision or Mixed-Precision Arithmetic 11
5.3 Control Flow 12
5.4 Data Memory 13
5.5 Function Call with Parameters on Stack, "Callee-Save" Registers 13
5.6 Function Call with Parameters in Registers, "Caller-Save" Registers 18
5.7 Other options 20

Bibliography 20

List of figures

Typical Stack Frame with Parameters and Return Value on Stack 14
Function Calls with Parameters on Stack 15
Function `two_x_plus_y` That was Called in Figure 2 16
Main Program to Call the Function `foo` in Figure 2 or Figure 6. 16
Stack Layout for Figures 2, 3, and 4 at Time of `RETURN` From `two_x_plus_y`. 17
Function Calls with Parameters on Stack and an Escaping Local Variable 18
Function `two_a_plus_y`, Which was Called in Figure 6. 16
Typical Stack Frame with Parameters and Return Value in Registers 19
Function Calls with Parameters in Registers 19
Function `two_x_plus_y` That was Called in Figure 9 20

1 Purpose and History

The Haverford Educational RISC Architecture (HERA) provides a foundation for a multi-course project unifying Haverford's upper-level computer science curriculum. HERA is powerful enough to introduce assembly language programming in Principles of Programming Languages and serve as a target for compilers in Compiler Design, yet be simple enough to be built as a student project in Principles of Computer Organization (using, for example, TKGate [Han04]) and extended in hardware/software co-design projects in Operating Systems. Thus, through these four classes, students produce a system in which high-level code is translated into machine language that can be executed on a microprocessor they have designed, and on which I/O to an ASCII terminal can be performed through simple device drivers they have written themselves. This philosophy of building the simplest system that actually works, is coupled with lectures contrasting HERA with real-world systems. For more details of the educational uses of HERA, see [Won06].

The HERA-C development system allows students to execute HERA assembly language programs before their own system is operational. HERA-C is a set of C macros that allows a standard C or C++ development environment to compile, execute, and debug HERA programs. This lets students start to use HERA with a minimum amount of distraction from new tools. See [Won03] for more information about HERA-C and other supporting tools.

The HERA system originated with an attempt to simplify 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!

2 Architecture Overview

The HERA processor has seventeen 16-bit registers, known as PC and $R_0 ... R_{15}$. All but the Program Counter (PC) can be used as operands for most instructions (e.g. `ADD`, `LOAD`, etc.). The Stack Pointer (R_{15}, also known as SP) and Frame Pointer (R_{14}, a.k.a. FP) are also modified by the function call and return operations. The temporary register (R_{13}, a.k.a. R_t or TMP) is used during function calls, returns, and in multiplication operations, as well as by some assembly-language pseudo-operations. The zero register (R_0) always has the value 0, providing a mechanism for performing comparison (the `CMP` pseudo-op), negation (`NEG`), and other things.

The processor status and control flags are: sign (s, or F_0), which is set to true by a negative result; zero (z, or F_1), which is set to true by a zero result; overflow (v, or F_2), which is set to true by an overflow from a 2's complement arithmetic operation; and carry (c, or F_3), which is set to true by an overflow from an unsigned operation. These flags may be used in a branch instruction, and the value of the carry flag may be used in subsequent arithmetic operations. There is an additional 5th flag F_4 known as "carry-block". When carry-block is set the carry is not used during arithmetic operations, providing for faster, simpler code for single-precision operations, or during shift operations. The carry-block flag can be saved, restored, or explicitly modified, but is not affected by other operations.

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 "Harvard" architecture[Wik07]).

3 Instruction Set

The HERA instruction set is comprised entirely of single-word operations. The vast majority of operations involve only registers.

3.1 Arithmetic, Shift, and Logical Instructions ($b_{15} = 1$, $b_{15:12} = 0011$)

The Arithmetic, Shift and Logical Instructions take three forms, one for operations involving immediate values, one for common arithmetic and bitwise operations (using three-address form), and one for other operations requiring fewer than three operands.

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

SETLO and SETHI operate on a destination register number and an immediate 8-bit value.

$b_{15:12}$	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, while `SETHI` changes the high 8 bits of R_d to v. A `SETLO`/`SETHI` sequence can thus be used to establish any arbitrary 16-bit pattern in a register, while `SETLO` by itself provides small constants. `SETLO` and `SETHI` do not affect any flags.

The format for the binary instructions for `SETLO` and `SETHI` is

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	l/h	d				v							

3.1.2 Three-address operations ($b_{15:13} = 110$, 101, and 100)

The three-address HERA operations are

$b_{15:12}$	Mnemonic	Meaning	Notes
1000	`AND`(d, a,b)	$R_d(i) \leftarrow R_a(i) \wedge R_b(i)$	bitwise logical and
1001	`OR`(d, a,b)	$R_d(i) \leftarrow R_a(i) \vee R_b(i)$	bitwise logical or
1010	`ADD`(d, a,b)	$R_d \leftarrow R_a + R_b + (c \wedge F_4')$	use carry unless blocked
1011	`SUB`(d, a,b)	$R_d \leftarrow R_a - R_b - (c' \wedge F_4')$	use carry unless blocked
1100	`MULT`(d, a,b)	$R_d \leftarrow (R_a * R_b)_{15:0}$, $R_t \leftarrow (R_a * R_b)_{31:16}$	*signed* multiplication
1101	`XOR`(d, a,b)	$R_d \leftarrow R_a \oplus R_b$	bitwise exclusive or

These operations all set the zero flag (to true if and only if the result of the operation was zero) and sign flag (true iff b_{15} of the result is true). Addition and subtraction set the overflow flag and carry flag. Unless the carry-block flag is set, addition and subtraction use the carry flag as the incoming carry. If carry-block is true, addition and subtraction ignore the carry flag (carry-in is 0 for addition and 1 for subtraction).

The multiplication operation computes the product of R_a and R_b, treated as *signed* integer quantities. It places the low-order 16 bits in R_d and the high-order 16 bits in the temporary register R_t. The zero flag is set only if all 32 bits of the result are 0, the sign flag is set if the result is negative, the overflow flag is set if simply sign extending the lower 16 bits would not produce the full 32-bit result, and the carry flag is not defined (software should not rely on any particular effect (or lack of effect) on the carry flag).

These operations are all encoded by the operation followed by d, a and b.

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

3.1.3 Shifts, increments, and flag operations ($b_{15:12} = 0011$)

When $b_{15:12} = 0011$, a shift, increment, or flag control operation is performed, based on $b_{7:4}$.

Shifts The HERA shift operations are:

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

Shift operations do not set any flags except carry, which is set (to the bit shifted out) only by the one-bit shift operations. `LSL`, `LSR`, and `ASL` shift in ($c \wedge F_4'$). Thus, when carry-block is false, the logical shift operations correspond to a rotate with carry. The eight-bit shift operations and `ASR` shift in zeros, regardless of the carry block.

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

Set/clear flags Flags (or sets of flags) can be explicitly set or cleared with `SETF` or `CLRF`.

$b_{15:12}$	b_{11}	$b_{7:4}$	Mnemonic	Meaning	Notes
0011	0	0110	`SETF`(v)	$F \leftarrow F \vee v$	Set flags for which v is true
0011	1	0110	`CLRF`(v)	$F \leftarrow F \wedge v'$	Clear flags for which v is true

The value of b_{11} controls whether flags are set ($b_{11} = 0$) or cleared ($b_{11} = 1$). The values in b_8 and $b_{3:0}$ are combined to make a five-bit value that is used to control which flags are set or cleared. The `SETF` and `CLRF` instructions are typically treated as single-operand instructions by assemblers, rather than operations with separate operands for v_4 and $v_{3:0}$. In other words, `SETF(0x15)` produces the instruction $0x\,3865$, which sets the carry block (F_4, or 0x10), overflow (F_2, or 0x04) and sign (F_0, or 0x01) flags, while `CLRF(0x0a)` produces 0x306a, which clears the carry (F_3, or 0x08) and zero (F_1, or 0x02) flags.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	s/c	0	0	v_4	0	1	1	0	v_3	v_2	v_1	v_0

Save/restore flags Flags can also 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

The value of b_3 controls whether flags are saved ($b_3 = 0$) or restored ($b_3 = 1$). Note that the flags are saved in bits 4:0 of R_d, not b_8 and $b_{3:0}$ as in the SETF and CLRF instructions.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1		d			0	1	1	1	s/r	0	0	0

Increments The increment and decrement operations are

$b_{15:12}$	$b_{7:6}$	Mnemonic	Meaning	Notes
0011	10	INC(d, δ+1)	$R_d \leftarrow R_d + \delta^\star$	Increment R_d by $\delta(\text{unsigned}(b_{5:0})) + 1$
0011	11	DEC(d, δ+1)	$R_d \leftarrow R_d - \delta^\star$	Decrement R_d by $\delta(\text{unsigned}(b_{5:0})) + 1$

The increment and decrement operations update the overflow and carry flags (as well as zero and sign), 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.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1		d			1	i/d			v			

Note that the value added or subtracted from R_d is one more than the unsigned quantity v — there is no increment or decrement by zero. Note that, 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.

3.2 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 sets the s and z flags. In addition to LOAD and STORE, the function CALL and RETURN instructions (see Section 3.3) also interact with memory. All other instructions affect only registers and flags.

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 s 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_{13} followed by $b_{7:4}$). By convention, assemblers combine all o bits into a single offset parameter, e.g. LOAD(r1, 17,r2) loads $M[R_2 + 17]$ into R_1.

3.3 Control-Flow and Other Instructions ($b_{15:14} = 00$)

Control-flow instructions include conditional branches, unconditional branches ("jump" instruction), function and interrupt instructions.

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

HERA provides the following branches that transfer to an address in a register (b) and relative branches (distinguished by an appended "R" in the assembly language operation) that transfer to the current positions plus an 8-bit signed offset (o).

$b_{15:12}$	$b_{11:8}$	Mnemonic	Meaning
0001/0	0000	BR(b)/BRR(o)	Unconditional branch – $true$
0001/0	0001		*(special branches – see below)*
0001/0	0010	BL(b)/BLR(o)	Branch if signed result <0 — $(s \oplus v)$
0001/0	0011	BGE(b)/BGER(o)	Branch if signed result $\geqslant 0$ — $(s \oplus v)'$
0001/0	0100	BLE(b)/BLER(o)	Branch if signed result $\leqslant 0$ — $((s \oplus v) \vee z)$
0001/0	0101	BG(b)/BGR(o)	Branch if signed result >0 — $((s \oplus v) \vee z)'$
0001/0	0110	BULE(b)/BULER(o)	Branch if unsigned result $\leqslant 0$ — $(c' \vee z)$
0001/0	0111	BUG(b)/BUGR(o)	Branch if unsigned result >0 — $(c' \vee z)'$
0001/0	1000	BZ(b)/BZR(o)	Branch if zero — z (if CMP operands $=$)
0001/0	1001	BNZ(b)/BNZR(o)	Branch if not zero — z' (if operands $\neq$)
0001/0	1010	BC(b)/BCR(o)	Branch if carry — c (unsigned result $\geqslant 0$)
0001/0	1011	BNC(b)/BNCR(o)	Branch if not carry — c' (unsigned <0)
0001/0	1100	BS(b)/BSR(o)	Branch if sign (negative) — s
0001/0	1101	BNS(b)/BNSR(o)	Branch if not sign (non-negative) — s'
0001/0	1110	BV(b)/BVR(o)	Branch if overflow — v
0001/0	1111	BNV(b)/BNVR(o)	Branch if not overflow — v'

Branches, both conditional and unconditional, are indicated by $b_{15:13} = 000$. For both, $b_{11:8}$ indicate the condition c under which the branch is to be taken; See section 9-8 of M. Morris Mano's "Computer Engineering: Hardware Design" for an explanation of which flags are used for each branch.

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		c			0	0	0	0		b		

where R_b gives the address for the next instruction to be executed if c is satisfied.

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		c						o				

where o is treated as an 8-bit *signed* quantity (-128...127) giving the offset from the branch instruction itself (i.e., the address for the next instruction to be executed if c is satisfied is $PC + o$, where PC indicates the value of the program counter for the branch instruction itself). Thus, the unconditional relative branch instruction ($b_{7:0} = 0$) includes the special cases HALT ($o = 0$) and NOP ($o = 1$).

Special branches Branches for which $b_{11:8}$ is 0001 are treated as special cases. The following are the current special-case branch instructions — all other branches with $b_{11:8} = 0001$ are undefined. The exact specification of `SWI`, `RTI`, and hardware interrupts is left as an exercise for our Operating Systems course.

b				Mnemonic	Comments
0001	0001	0000	i	`SWI`(i)	Software interrupt $\#i$
0001	0001	0001	0000	`RTI`()	Return from interrupt
0001	0001	0001	0001	`RETURN`()	Return from function call

3.3.2 Function calls ($b_{15:12} = 0010$) and returns (0001 0001 0001 0001)

The HERA function call instruction is

$b_{15:12}$	Mnemonic	Comments
0010	`CALL`(s,b)	Call function at R_b with initial stack frame size s

where R_b gives the starting address of the function, and s is an 8-bit *unsigned* quantity giving the size of the initial stack frame for the called function (0...255 words). The function call instruction saves the address of the next instruction (i.e. the target of the `RETURN` instruction) on the stack, updates the program counter to R_b, updates the stack pointer and frame pointers, and saves the old frame pointer in R_t:

$$M[SP] \leftarrow PC + 1, PC \leftarrow R_b, R_t \leftarrow FP, FP \leftarrow SP, SP \leftarrow SP + s$$

The `RETURN` instruction (see Section 3.3.1 above) reverses a `CALL` instruction, i.e.,

$$PC \leftarrow M[FP], FP \leftarrow R_t, SP \leftarrow FP$$

Note the importance of restoring R_t if it has been modified during the function by an operation such as `MULT` or `CALL`, by a pseudo-op, or by explicit use as the result register for another operation.

The function call instruction has the binary format

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

4 Assembly Language Conventions and Pseudo-Operations

Operands for the operations listed in the previous section are listed in the order they appear in the instruction (from b_{15} to b_0). For example, `AND(r1, r2,r3)` sets R_1 to $R_2 \wedge R_3$, and `STORE(r1, 5,r2)` puts the contents of R_1 into $M[R_2 + 5]$. Note again that `INC` and `DEC` require the programmer to express the quantity to be added or subtracted, i.e. `INC(r1,6)` produces a machine-language instruction containing the operand 5 (i.e. $0xd185$), which adds 6 to R_1.

HERA assembly language typically defines the following pseudo-operations in addition to the true operations listed in the previous section. NOTE that the HERA-C simulator requires that all data and data labels (i.e., `INTEGER`, `TIGER_STRING`, and `DLABEL`) precede any instructions.

Mnemonic	Definition	Notes
`SET`(d, l)	`SETLO`(d, l&0$x$ff); `SETHI`(d, $l \gg 8$)	$R_d \leftarrow l$ (set R_d to 16-bit value l)
`CMP`(a, b)	`SETC`( ); `SUB`($R0$, a, b)	Set flags for $a - b$
`NEG`(d, b)	`SETC`( ); `SUB`(d, $R0$, b)	Set $R_d \leftarrow - R_b$
`NOT`(d, b)	`SET`(R_t, 0x ffff); `XOR`(d, R_t, b)	Bitwise complement
`HALT`( )	`BRR`(0)	Halt the program
`NOP`( )	`BRR`(1)	Do nothing ("No operation")
`SETC`( )	`SETF`(0x08)	Set the carry flag
`CLRC`( )	`CLRF`(0x08)	Clear the carry flag
`SETCB`( )	`SETF`(0x10)	Set the carry-block flag
`CLCCB`( )	`CLRF`(0x18)	Clear carry and carry-block flags
`FLAGS`(a)	`CLRC`( ); `ADD`($R0$, a, $R0$)	Set flags for R_a
`LABEL`(L)/`DLABEL`(L)	*(no machine language generated)*	Define a label $L^\star$
`INTEGER`(i)	i	Put i in the current memory cell
`TIGER_STRING`(s)	$s^{\star\star\star}$	Put string s in memory for Tiger
`CALL`(s, L)	`SET`(R_t, address(L)); `CALL`(s, R_t)	Do a call using R_t (i.e., R_{13})
`BR`(L)	`SET`(R_t, address(L)); `BR`(R_t)	Do a branch to label L using $R_t^{\star\star}$
`BRR`(L)	`BRR`(address(L))	Do a relative branch to label L
`BG`(L), `BC`(L)...	...	Do any other branch using $R_t^{\star\star}$
`BGR`(L), `BCR`(L)...	...	Do any other relative branch

* A label is a sequence of letters, numerals, and underscores starting with a letter (i.e., any C++ identifier *that does not start with an underscore*). A label may be used in a `CALL` or branch instruction, or to identify the address of subsequent values stored with `INTEGER` and `TIGER_STRING` pseudo-ops. Due to a limitation in its implementation, HERA-C simulator requires the separate `DLABEL` pseudo-op for labeling data rather than instructions.

** By convention, when a register-mode branch is used with a label, R_t is used.

*** The string s cannot contain control characters (including newlines or tabs) or the backslash — any string that must contain these should be created as a sequence of `INTEGER` pseudo-ops giving the ASCII values. Some assemblers *may* accept these characters or choose to interpret sequences starting with backslash as a C compiler would, but this usage may not be portable.

5 Idioms

The following sections give typical usages of HERA operations and pseudo-operations.

5.1 Single-Precision Arithmetic

If a program employs only single-precision addition and subtraction, it typically begins by setting the carry-block flag, and then uses whatever arithmetic operations it requires, like so:

```
// Set R1 to the single-precision sum of R2+R3+R4, R5 to R4-R3

SETCB()        // Disable use of carry flag in single-precision
  //  ...          (other instrucitons that may set the carry; we don't care)
ADD(r1, r2,r3) // R1 = R2 + R3,             * regardless of incoming carry *
ADD(r1, r1,r4) // R1 = R1 (i.e. R2+R3) + R4, * regardless of carry *
SUB(r5, r4,r3) // R5 = R4 - R3,             * regardless of carry *
```

Or, in machine language, `3160 ... a123 a114 b543`.

5.2 Double-Precision or Mixed-Precision Arithmetic

If a program employs double (or higher) precision arithmetic, i.e. uses two or more 16-bit registers to represent a value of 32 (or more) bits, then the carry block must be disabled (or left off if the processor has been reset before the program starts). Programs that employ a mixture of single and higher precision may either turn carry-block on and off, or leave carry-block off and set the carry flag before single as well as double-precision operations. When carry-block is off, the carry flag should generally be *cleared* before addition and *set* before subtraction. For example:

```
// Make [R1 R2] the sum of [R3 R4], [R5 R6], and [R7 R8]), then R9=-R9

CLCCB()        // Enable use of carry flag for multiple-precision arithmetic
  // ...          (possibly other instructions that may set or clear the carry)
CLRC()         // Start with carry-in=0 for least-significant digit
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
CLRC()         // Start with carry-in=0 for least-significant digit of [R7 R8]
ADD(r2, r2,r8) // R2 = R2+R8 (i.e. R4+R6+R8), carry set if necessary
ADD(r1, r1,r7) // R1 = R1+R7, i.e. the most significant digit of the sum
SETC()         // Make sure carry-in=1 for subtraction
SUB(r9, r0,r9) // Set R9=-R9 (as in the pseudo-op NEG(r9))
```

Or, in machine language, `3968 ... 3868 a246 a135 3868 a228 a117 3068 b909`.

5.3 Control Flow

Control flow normally proceeds through consecutive instructions (moving from address a to address $a + 1$ of the instruction memory). Branches (and `CALL` and `RETURN` — see Sections 5.5-5.7) provide a mechanism for moving to another address instead.

After executing a register-mode branch `BR(`R_b`)`, a HERA processor will next execute whatever instruction is in the instruction memory cell specified by R_b. So, for example the sequence `SET(Rt, 0x0174) BR(Rt)` (`ed01 fd74 100d` in machine language), the processor will execute whatever instruction are in instruction memory cells $0x0174$, then $0x0175$, then $0x0176$, etc. (unless $0x0174$ or $0x0175$ contains a branch, `CALL`, or `RETURN`).

After executing a relative-mode branches `BRR(`o`)` in instruction address a, a HERA processor will next execute the instruction in address $a + o$. Thus, the processor would execute the instruction in address $0x0174$ after a `BRR(4)` (`0004`) in address $0x170$ or a `BRR(-4)` (`00fc`) in address $0x0178$. Note that `BRR(0)` (`0000`) prevents the processor from moving on to the next instruction, and `BRR(1)` (`0001`) does nothing but go on to the next instruction.

Branch targets are defined in assembly language with `LABEL`s. *Note* that assemblers generally translate long-distance branch instructions as a `SET(Rt, ...)/BR` sequence, and they may also treat short-distance branches this way, so it is not safe to rely on the value in the temporary register (R_{13}, also known as R_t) after a branch instruction. The HERA-C simulator overwrites R_t as part of each branch instructions, to discourage the habit of relying on R_t inappropriately.

Conditional control flow is expressed as on most other modern microprocessors — arithmetic operations are used to establish values of the flag bits, and these flags then control the action of conditional branch instructions such as `BZ`, `BZR`, `BLE`, `BLER`, etc.. For example, the program below 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 (i.e., if $(s \oplus v)$ is false, indicating that the sign flag is false (for a positive or zero result) without an overflow, or that sign is true in the presence of overflow). Thus the negation of the value in R_1 is skipped if R_1 is already positive.

```
// Make R2 = (abs(r1))/2 (using r1=-74, for example). In other words:
//   if r1 < 0: r1 = 0-r1
//   r2 = r1/2

SETCB()         // Use single-precision
SETLO(r1, 0xB6) // Try -74 for this example

CMP(r1, r0)     // set flags for r1-0, i.e. r1
BGER(SKIP_NEGATION) // if flags show result >= 0, skip over SUB
SUB(r1, r0,r1)
LABEL(SKIP_NEGATION)
LSR(r2, r1)     // divide by 2 via logical shift right
```

An assembler would typically produce the sequence `3160 e1b6 3818 b010 0302 b101 3211` 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 $0x0208$ (the address to be used for the `LSR`) into R_{13}, producing the sequence `3160 e1b6 3818 b010 ed08 fd02 130d b101 3211`.

5.4 Data Memory

Values can be placed in data memory via `INTEGER` (for integers) or `TIGER_STRING` (for strings of characters preceded by a character count, as in Andrew Appel's "Tiger" language or most Pascal systems). Locations of these values can be identified via `DLABEL`, allowing the use of `SET` to establish an address in a register, followed by `LOAD` or `STORE` to retrieve or store a value at that address. The following program computes the sum of the absolute values of the integers placed in memory from the location labelled `where_A_starts` until the first location containing zero (five cells later, in this example), and places this sum in the memory cell labelled `where_B_is`.

```
DLABEL(where_A_starts)
  INTEGER(31)
  INTEGER(33)
  INTEGER(-37)
  INTEGER(41)
  INTEGER(-43)
  INTEGER(0)
DLABEL(where_B_is)
  INTEGER(0)  // B GOES HERE

  SETCB()
  SET(R1, where_A_starts)
  SET(R2, 0)
LABEL(top_of_loop)
  LOAD(R3, 0,R1)
  CMP(R3, R0)
BZ(done_with_sum)
BG(no_need_to_negate)
  SUB(R3, R0,R3)
LABEL(no_need_to_negate)
  ADD(R2, R2,R3)
  INC(R1, 1)
BR(top_of_loop)
LABEL(done_with_sum)
  SET(R1, where_B_is)
  STORE(R2, 0,R1)
```

If this program were assembled by itself, the assembler might put the data values into data memory starting at address 0, and the instructions into instruction memory starting with its address 0, producing the sequence

```
0031 0033 .... .... .... 0000 0000
```

for the data memory and

```
3160 e100 f100 4301 3068 b030 ed.. fd00 180d ed.. fd00 150d
b303 a223 3180 ed05 fd00 100d e106 f100 6201
```

for instruction memory.

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

5.5 Function Call with Parameters on Stack, "Callee-Save" Registers

The HERA CALL and RETURN instructions automatically update the Frame Pointer (FP or R_{14}) and Stack Pointer (SP or R_{15}), and CALL saves the return address in the memory location pointed to by the new Frame Pointer. The programmer or compiler must generate other instructions to establish a "static link" (if used) and manage the "control link" (the value of the calling function's FP, which the CALL instruction places in R_t and the RETURN instruction expects to find there). This is typically done with SL in memory cell $FP+1$ and CL in $FP+2$, if a static link is used, or just with CL in $FP+1$ if not. There are several approaches to handling the values of function parameters, local variables, and temporaries, and also to solving the problem of what to do with the values in registers when one function calls another. These are discussed briefly in this and the following section; most programming language or compiler textbooks provide more details on function call implementation, for example see [App98].

A simple and general approach to passing parameters to functions is to place their values in consecutive memory cells of the stack frame of the function to be called. The function can then increase the size of the stack frame (by incrementing SP beyond the initial value generated by the CALL) to allow for local variables and temporary values. A simple way to avoid problems in which a called function overwrites a register used by the calling function is to have every function start by saving all registers it will use and then re-load these values just before it returns. Figure 1 shows a typical stack frame layout for this approach (space for local variables, temporaries, and saved registers is referred to as *local storage* in the figure and below).

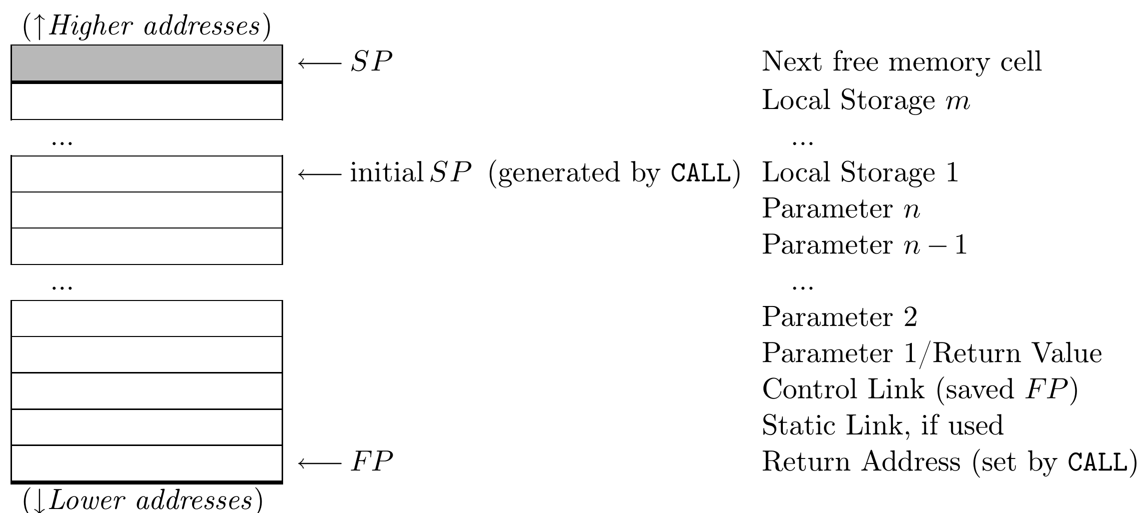

Figure 1. Typical Stack Frame with Parameters and Return Value on Stack

To call a function using these conventions, we

- Put the parameters on the stack starting at $SP+3$ (this will be the callee's $FP+3$)
- Set up the static link in cell $SP+1$
- Issue the CALL instruction, with initial frame size s being the number of parameters $+3$
- After the call is completed, the returned value can be retrieved from $SP+3$.

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

- Increment SP to make space for local storage (from "initial SP" to "SP" in Fig. 1)
- Save the old FP value (from R_t) and any other registers this function will use

- Give the function body (in which we load parameters from the stack frame, e.g. $FP+3$)
- Store the return value at $FP+3$
- Restore saved registers, including the old frame pointer value back into R_t
- `RETURN` from the function

Figure 2 shows an example function in HERA assembly language, using the conventions of Figure 1. Note that, in this and the following function call examples, we assume the carry block flag has been set to allow single-precision arithmetic without explicit clearing or setting of the carry flag. The function `foo` expects two parameters (in memory cells $FP + 3$ and $FP + 4$) and calls `two_x_plus_y`, which also expects two parameters (in the same positions in *its* stack frame). The `foo` function begins by increasing its stack frame size by 2, to make space in which to save the

```
// 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 them and old FP (Rt)
    INC(SP, 2)
    STORE(Rt, 2,FP)   // No static link used here, but skip FP+1 for uniformity
    STORE(r1, 5,FP)   // skip FP+3 and FP+4, where a and b will be
    STORE(r2, 6,FP)

// set up parameters a+b and b-a+75 for the call to two_x_plus_y
    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,SP)   //  1st parameter at SP+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,SP)   //  establish 2nd parameter at SP+4

// Build the static link for two_a_plus_y (points to foo's frame), do the call
    STORE(FP, 1,SP)
    CALL(5,two_x_plus_y) // initial stack frame size 5 (RA, SL, Old FP, 2 params)

// Now retrieve result and multiply by "a"
    LOAD(r2, 3,SP)    // R2 = result retured FROM CALLED FUNC'S FRAME
                      // NOTE that r1 is still "a" from before the call
    MULT(r1, r2,r1)

// Save result, restore registers (including Rt to old FP) and return
    STORE(r1, 3,FP)   // Put return value over 1st parameter
    LOAD(r2, 6,FP)    // Restore r2
    LOAD(r1, 5,FP)    // Restore r1
    LOAD(Rt, 2,FP)    // Restore Rt
    RETURN()
```

Figure 2. Function Calls with Parameters on Stack

two registers it will use. Note that it does this using INC, which does not change any other registers. It then saves the old frame pointer (R_t) at $FP+2$ and saves the other registers it will use in the spaces it allocated with the INC ($FP+5$ and $FP+6$). After this "preamble", foo begins the work to compute its result. During its operation, foo makes use of the parameters a and b by loading them into registers from $FP+3$ and $FP+4$, and establishes the values of the parameters x and y for two_x_plus_y in memory cells $SP+3$ and $FP+4$. After the call to two_x_plus_y, foo retrieves the returned value from $SP+3$. Note it can still rely on the value of a that it left in R_1 before making the call. Upon completing its work, foo stores its result in memory cell $FP + 3$ and follows a standard "postamble" sequence of restoring the registers saved in the preamble and RETURNing.

```
// 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)
    INC(SP, 2)
    STORE(Rt, 2,FP)   // SL not actually used in this example...
    STORE(r1, 5,FP)
    STORE(r2, 6,FP)

// Load "x" and "y" from stack frame
    LOAD(r1, 3,FP)   // R1 = x
    LOAD(r2, 4,FP)   // R2 = y

// Compute the result
    ADD(r1, r1,r1)       // result = 2*x
    ADD(r1, r1,r2)       // result = 2*x+y

// Store the result, restore registers, and return
    STORE(r1, 3,FP)
    LOAD(r2,  6,FP)
    LOAD(r1,  5,FP)
    LOAD(Rt, 2,FP)    // Restore Rt (not needed in this function)
    RETURN()
```

Figure 3. Function two_x_plus_y That was Called in Figure 2

To complete this example, Figure 3 shows the function two_x_plus_y, Figure 4 gives a main program to call foo with the parameters 10 and 2, and Figure 5 shows a digram of the stack generated by running this program (at the instant just before the RETURN from two_x_plus_y; 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). Note that two_x_plus_y contains a preamble/postamble similar to those of foo. Since two_x_plus_y doesn't overwrite R_t (either explicitly or via a CALL, MULT, or register-mode branches generated by the assembler), it doesn't have to save and restore its control link (the caller's FP), but this code is included here as part of the standard convention. Note also that the main program is not a called function, and thus does not need a return address, static link, or dynamic link. Furthermore, the example in Figure 4 does not require any local storage. Thus, it essentially has a frame of size 0, with $FP = SP = 0$, as shown in Figure 5.

```
// Main program to call "foo(10, 2)"
// Assumes HERA has just been started up:
//  All flags (including carry block) are FALSE,
//  All registers (including SP and FP) are 0.

SETCB()
SET(r1,10)
STORE(r1, 3,SP)
SET(r1,2)
STORE(r1, 4,SP)
CALL(5,foo)
LOAD(r1,  3,SP) // load result of foo into R1
```

Figure 4. Main Program to Call the Function `foo` in Figure 2 or Figure 6.

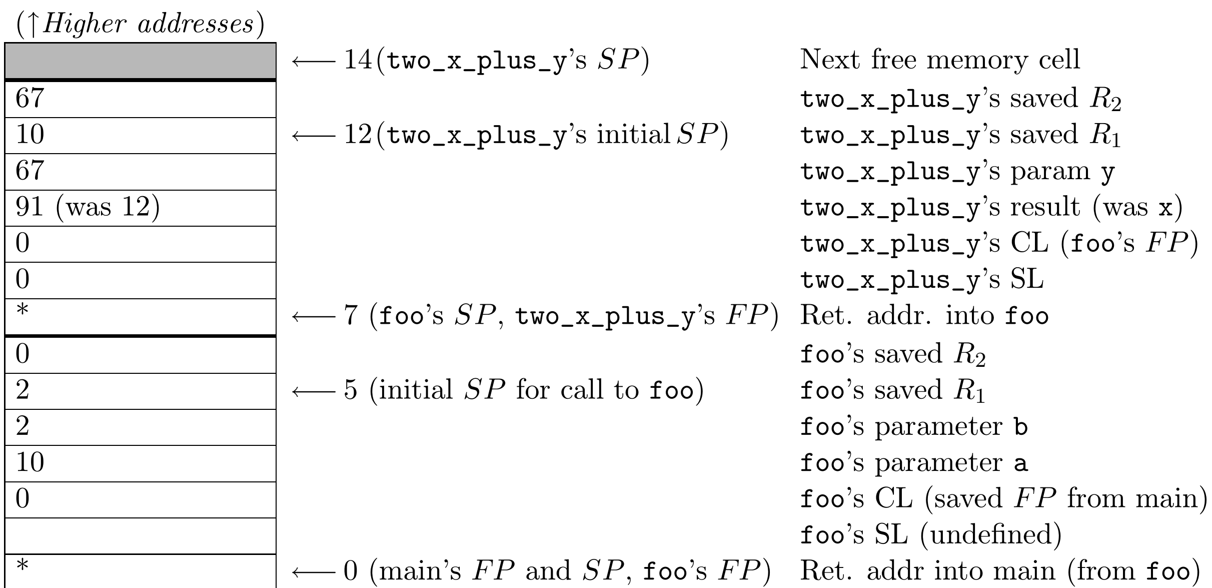

Figure 5. Stack Layout for Figures 2, 3, and 4 at Time of `RETURN` From `two_x_plus_y`.

The `two_x_plus_y` example does not illustrate the use of a static link. Figures 6 and 7 show a variant on this example in which the local variable `a` escapes from `foo` into `two_a_plus_y`. The static link for `two_a_plus_y` is established right before the call, at offset 1 in the stack frame under construction. This allows `two_a_plus_y` to find the value of `foo`'s variable `a` by retrieving `foo`'s frame pointer and loading the variable at offset 3.

5.6 Function Call with Parameters in Registers, "Caller-Save" Registers

If no parameter ever "escapes" from a stack frame, no static link is needed; if this is true and all functions take eleven or fewer parameters, all parameters can be passed in $R_1 \ldots R_{11}$, without using the stack for parameters. When this convention is used, the registers are often saved by the calling function, rather than the called function.

```
// Translation of function foo (single precision, assuming carry-block is set)
//     int foo(int a, int b) =
//          let two_a_plus_y(y : int): int = a+a+y
//          in  two_a_plus_y(b-a+75) * a

LABEL(foo)
// FIRST, make space for old FP and save it and registers we'll overwrite
    INC(SP, 2)
    STORE(Rt, 2,FP)   // put Rt at FP+2, leaving space for static link at FP+1
    STORE(r1, 5,FP)   // Store R1 at FP+5, etc.
    STORE(r2, 6,FP)

// Now build the parameter (b-a+75) for two_a_plus_y (putting it in SP+3)
    LOAD(r1, 3,FP)    // R1 = a (FP+3)
    LOAD(r2, 4,FP)    // R2 = b
    SUB(r2, r1,r2)    // R2 = a-b
    SETLO(Rt, 75)
    ADD(r2, r2,Rt)   // R1 = a-b+75
    STORE(r2, 3,SP)

// Build the static link for two_a_plus_y (points to foo's frame), do the call
    STORE(FP, 1,SP)
    CALL(4,two_a_plus_y)

// Now multiply the result by "a", which will still be in R1
    LOAD(r2, 3,SP)    // R2 = result
    MULT(r1, r1,r2)

// Save result, restore registers and return
    STORE(r1, 3,FP)
    LOAD(r2, 6,FP)
    LOAD(r1, 5,FP)
    LOAD(Rt, 2,FP)
    RETURN()
```

Figure 6. Function Calls with Parameters on Stack and an Escaping Local Variable

```
// Translation of two_a_plus_y with lexically scoped y
LABEL(two_a_plus_y)
    INC(SP, 2)
    // don't bother saving Rt since we don't overwrite it
    STORE(r1, 4,FP)
    STORE(r2, 5,FP)

// Load "a" from offset 3 of statically scoped frame, then "y" from this frame
    LOAD(r1, 1,FP)   // get two_a_plus_y's static link, i.e., foo's FP
    LOAD(r1, 3,r1)   // r1 now is a
    LOAD(r2, 3,FP)   // r2 = y

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

// Store the result, restore registers, and return
    STORE(r1, 3,FP)
    LOAD(r2, 5,FP)
    LOAD(r1, 4,FP)
    RETURN()
```

Figure 7. Function `two_a_plus_y`, Which was Called in Figure 6.

The general stack frame layout for this approach is shown in Figure 8. With this convention, functions still use the stack for return addresses, because of the HERA `CALL` instruction. Functions also use the stack for any local variables or temporary values that don't fit in registers, and to save register values before making a call. To call a function using these conventions, we

- Save (on the stack) any registers we may need after the call
- Put the parameter values into the appropriate registers
- Issue the `CALL` instruction (after the call, the returned value can be retrieved from R_1)

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

- Increment the SP to make space for any local storage that is needed
- If the function might overwrite the old FP in R_t, save it somewhere
- Give the function body, putting its result into R_1
- Restore the old FP value into R_t and `RETURN` from the function.

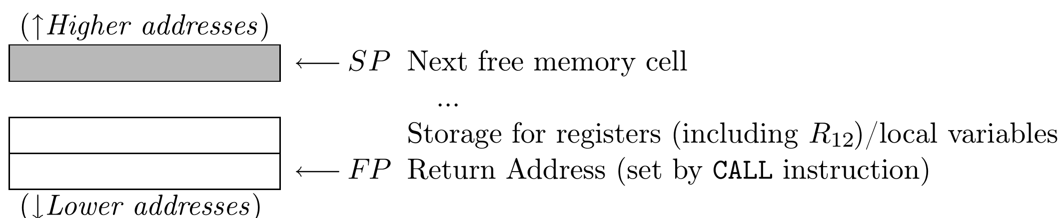

Figure 8. Typical Stack Frame with Parameters and Return Value in Registers

Figures 9 and 10 show our `two_x_plus_y` example written to use the “parameters in registers; no static link; caller-save registers” conventions.

```
// Translation of foo (single precision, assuming CB set, no static links used)
//     int foo(a : int, b : int) : int = two_x_plus_y(a+b, b-a+75) * a

LABEL(foo)
// FIRST, make space to save old FP and a (since we'll need a after the call)
    INC(SP, 2)
    STORE(Rt, 1,FP)         // No static link used, put old FP (Rt) in FP+1
// get "a" and "b" out of r1 and r2 so that we can use r1 and r2 in next call:
    ADD(r10, r1,r0)
    ADD(r11, r2,r0)

// 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
// save "a" (needed after the call)
    STORE(r10, 2,FP)

// actually make the call
    CALL(1,two_x_plus_y)  // initial stack frame has size 1 for just return addr.

// Now restore "a" and multiply result of call (now in r1) by it:
    LOAD(r2, 2,FP)
    MULT(r1, r1,r2)

// Finally, restore Rt and do the return (result is already in r1)
    LOAD(Rt, 1,FP)
    RETURN()
```

Figure 9. Function Calls with Parameters in Registers

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

Figure 10. Function `two_x_plus_y` That was Called in Figure 9

5.7 Other options

Function call conventions can be combined in a variety of other ways. The choice among possible conventions must be made with respect to the programming language being used and degree of analysis/optimiation performed by the compiler or programmer, as well as the target architecture. For more information about other ways to combine the presence/absence of a static link, the use of stack or registers for parameters, and register saving by calling/called function, refer to a compiler implementation textbook.

Bibliography

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

[Han04] Jeffery P. Hansen. TKGate, a graphical editor and event-driven simulator for digital circuits with a tcl/tk-based interface. `http://www.tkgate.org/`, 1987-2004.

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

[Wik07] Wikimedia Foundation, Inc. Harvard architecture. `http://en.wikipedia.org/wiki/Harvard_architecture`, March 2007.

[Won03] David G. Wonnacott. HERA: The Haverford Educational RISC Architecture. `http:www.cs.haverford.edu/software/HERA/`, 2003.

[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.