A Reference Manual For a CISC-like Architecture Embedded in C

Mayer Goldberg

January 4, 2015

Contents

1 Introduction

This project is an *embedding* of a general-purpose, CISC-like micro-architecture into C. This project was written in support of the senior course in *Compiler Construction*, in the Department of Computer Science at Ben-Gurion University, Beer Sheva, Israel. The context of the project is the lack of a uniform common background in micro-architecture among the different students who are taking the *Compiler Construction* course. BGU offer several parallel tracks for studying micro-architecture, and these tracks diverge on course content, choice of architectures studied, choice of toolset chain, and level of programming expected. When students in these various tracks converge to the *Compiler Construction* course, they do not share a common experience in micro-architecture. The goals of this project are as follows:

- Expose the studens to a general CISC-like micro-architecture, with an orthogonal instruction set, and plenty of general-purpose registers.
- Allow for in-house experimentation & future development of the architecture (e.g., adding some SIMD, or vector-processing capabilities).
- Provide a common platform for exercises that involve code generation.
- Keep the toolset chain to a minimum: Stay within the familiar realm of *gcc*, *gdb*.
- Avoid issues of interfacing assembly language and C code. Simplify debugging by permitting a casual intermix of C and assembly.

2 Description of the Architecture

2.1 Registers

The architecture includes 16 general-purpose registers, `R0` through `R15`, that can store signed integers. The word size depends on the host computer, and is the same as the size of *long* in *gcc* on the host computer. By convention, the register `R0` is reserved for the return value of a procedure call. Additionally, there is a stack pointer `SP` and a frame pointer `FP`. The stack pointer is related to traditional instructions that are related to the stack: `PUSH`, `POP`, `CALL`, and `RETURN`. The frame pointer is just another general-purpose register. The frame pointer is meant to be used to provide a constant point of reference in the stack frame, during a procedure call.

Being an *embedded* architecture, there is no support for an instruction pointer register, and the user cannot write self-modifying code, or generate code at run-time.

2.2 Memory

For simplicity, memory is addressible at the word-size level (as opposed to the byte-size level). The amount of memory is controlled by the macro `RAM_SIZE` defined in the file `cisc.h`.

2.3 Operands & addressing

2.3.1 The structure of the stack

The top of the stack	
	...
`LOCAL(3)`	*local variable* 3
`LOCAL(2)`	*local variable* 2
`LOCAL(1)`	*local variable* 1
`LOCAL(0)` $\equiv$ `FPARG(-3)`	*local variable* 0 $\Leftarrow$ *current* `FP`
`FPARG(-2)`	*old fp*
`FPARG(-1)`	*ret addr*
`FPARG(0)`	*procedure argument* ARG_0
`FPARG(1)`	*procedure argument* ARG_1
`FPARG(2)`	*procedure argument* ARG_2
`FPARG(3)`	*procedure argument* ARG_3
	...
The bottom of the stack	

2.3.2 Addressing modes

Immediate An immediate operand is a constant passed onto an instruction. It is noted using the `IMM` prefix. For example, the following instruction moves the constant `5` to register: `R0`, you can do: `MOV(R0, IMM(5));` An immediate operand must be a number or a character. For example, the following instruction prints the letter `A`: `OUT(IMM(2), IMM('A'));`

Indirect An indirect operand refers to the address, in RAM, of some data. It is noted using the `IND` prefix. For example, suppose `R1` contains the *address* of a word. To load *that* word into `R0`, we can issue the following instruction: `MOV(R0, IND(R1));` Indirect references are suitable for accessing elements of *arrays*, *vectors*, and other data structures.

Indirect with displacement An indirect operand with a *displacement* is a convenient way of referring to an element of an *array*. While this can be accomplished by using the `IND` prefix, it must be preceeded by an `ADD` instruction in order to compute the correct displacement. A more concise & convenient way of doing the same is to use the `INDD` prefix. For example, `MOV(INDD(R0, 3), IMM(5))` moves the constant 5 to `R0[3]`.

Address

Referencing the stack Since the stack array memory is separate from the data array memory, the prefix `STACK` can be used to access the n-th element of the stack array.

Stack argument The prefix `STARG` is provided to access directly the 0-based, n-th argument from the stack pointer. While any position on the stack can be accessed directly and explicitly, it is safer and more concise to use the stack pointer or frame pointer. For example, after the instruction `CALL(FOO);` has executed, upon entry into the subroutine `FOO`, `STARG(-2)` is the next available position on the stack, `STARG(-1)` is the return address, `STARG(0)` is the 0-argument to the subroutine, `STARG(1)` is the 1-st argument, etc. Notice that `STARG` has a *shifted point of reference*, so as to match the corresponding argument to the subroutine.

Frame pointer argument The prefix `FPARG` is provied to access directly the n-th argument from the frame pointer. Keep in mind that the frame pointer needs to be set up by the programmer. Assuming the sequence `PUSH(FP); MOV(FP, SP);` is executed upon entry into a subroutine, then `FPARG(-3)` is the next available position on the stack, `FPARG(-2)` is the old frame pointer, `FPARG(-1)` is the return address, `FPARG(0)` is the 0-th argument to the subroutine, `FPARG(1)` is the 1-st argument, etc. Notice that, similarly to `STARG`, `FPARG` has a *shifted point of reference*, so as to match the corresponding argument to the subroutine.

Local variable The prefix `LOCAL` is provided to access directly the n-th local variable from the frame pointer. Keep in mind that the frame pointer needs to be set up by the programmer. Assuming the sequence `PUSH(FP); MOV(FP, SP);` is executed upon entry into a subroutine, then `LOCAL(0)` is the first local variable pushed onto the stack, `LOCAL(1)` is the second local variable pushed on to the stack, etc.

Label The prefix `LABEL` is provided to obtain the *address* of a label. The significance of this prefix comes from the fact that the micro-architecture is embedded in C: In standard C, labels are *symbolic* objects and the user cannot access the address in memory to which they refer. The GNU C compiler extends the ANSI C standard, to allow the programmer to obtain the address of a label. The assembly language inherits this pecularity, and treats labels as symbolic place holders. If you need to obtain an address for a label `L`, use `LABEL(L)`.

2.3.3 Labels

Labels are written as according to the syntax of C. If you wish to declare a block of local labels, you can write your code within a block:

```
BEGIN_LOCAL_LABELS L1, L2, L3;
...
L1:
...
L2:
...
L3:
...
END_LOCAL_LABELS;
```

These labels `L1`, `L2`, and `L3` are defined *only* within the local labels block. * The instruction set

2.4 Arithmetic instruction

`ADD(dest, src)` Add the source to the desination

`DECR(dest)` Decrement the destination

`DIV(dest, src)` Divide the destination by the source

`INCR(dest)` Increment the destination

`MUL(dest, src)` Multiply the destination by the source

`REM(dest, src)` Reduce the destination *modulo* the source

`SUB(dest, src)` Subtract the source from the desination

2.5 Logical instructions

`AND(dest, src)` The destination gets the *conjunction* with the source

`NEG(dest)` The destination is bitwise-complemented

`OR(dest, src)` The destination gets the *disjunction* with the source

`SHL(dest, src)` The destination is shifted to the *left* by as many bits as the source

`SHR(dest, src)` The destination is shifted to the *right* by as many bits as the source

`XOR(dest, src)` The desination is *xor*-ed with the source

2.6 Stack

`DROP(count)` Pops *count* elements off the top of the stack. These values are lost

`POP(dest)` Pop the top of the stack into the destination

`PUSH(src)` Push the source onto the top of the stack, advancing the `SP` register.

2.7 Comparison

`CMP(op1, op2)` Compares operands `op1` and `op2`. The special-purpose register `test_result` is assigned the difference of these two operands, and this quantity is used for *conditional jump* instructions (see section on *Control*).

2.8 Control

`CALL(dest)` Calls a procedure, where the target is specified as a label. The return address is pushed onto the stack. Any procedure arguments should be pushed onto the stack prior to the `CALL` instruction.

`CALLA(addr)` Calls a procedure, where the target is specified as an address. The return address is pushed onto the stack. Any procedure arguments should be pushed onto the stack prior to the `CALLA` instruction.

`HALT` Stops the micro-processor and exits.

`JUMP(dest)` An unconditional branch.

`JUMP_EQ(dest)` A branch conditional upon the result of a preceeding test being equal.

`JUMP_GE(dest)` A branch conditional upon the result of a preceeding test being greater than or equal.

`JUMP_GT(dest)` A branch conditional upon the result of a preceeding test being greater than.

`JUMP_LE(dest)` A branch conditional upon the result of a preceeding test being less than or equal.

`JUMP_LT(dest)` A branch conditional upon the result of a preceeding test being less than.

`JUMP_NE(dest)` A branch conditional upon the result of a preceeding test being not equal.

`RETURN` Return from a procedure call. The return address is popped off of the stack and used as the target of a jump. **Does not remove any arguments that were pushed onto the stack prior to the `CALL`.**

2.9 IO

`IN(dest, port)` Reads in a long value from the input port `port` and deposits it in `dest`.

`OUT(port, src)` Writes a long value from `src` to output port `port`.

At this time, only two ports are implemented:

- Port 1 for console input (from the keyboard)
- Port 2 for console output (to *stdout*)

2.10 Miscellaneous

`NOP` "No Operation". This instruction does nothing.* How to program in the micro-architecture

3 How to configure the micro-architecture

The embedded micro-architecture is implemented as a single C header file, `cisc.h`. If you are willing to get into the code, you can modify or add instructions rather easily. Otherwise, you should limit configuration to two macros:

```
#define RAM_SIZE Mega(1)
```

Change this value to modify the size of RAM.

```
#define STACK_SIZE Mega(1)
```

Change this value to modify the size of the stack.

The stack memory is not implemented as a part of the RAM, so these two macros are independent of each other.

4 Libraries

Library functions take their arguments from the stack, and return their [single] result to `R0`. Each function saves and restores all the registers it uses, except for `R0`, so that calling a library function should not change the contents of any register other than `R0`. When the function is not intended to return a value, upon return, the value of `R0` shall be unspecified.

4.1 The I/O library: `io.lib`

`GETCHAR` Reads a char from *stdin* and place it in `R0`.

`NEWLINE` Prints a newline character to *stdout*.

`PUTCHAR` Writes its char argument to *stdout*.

`READLINE` Returns a pointer to a dynamically-allocated string of chars, read from *stdin*, up to the end-of-line or the end-of-file.

`TAB` Prints a tab character to *stdout*.

`WRITE` Takes a pointer to a null-terminated string, and prints it to *stdout*.

`WRITE_INTEGER` Writes its integer argument to *stdout*.

`WRITELN` Takes a pointer to a null-terminated string, and prints it to *stdout*, followed by a *newline* character.

4.2 The mathematical library: `math.lib`

`ABS` Computes the absolute value of its argument: $R_0 \leftarrow |\mathrm{ARG}_0|$

`ACK` Compute Ackermann's function: $R_0 \leftarrow \mathrm{Ack}(\mathrm{ARG}_0, \mathrm{ARG}_1)$

`FACT` Compute the factorial function *recursively*: $R_0 \leftarrow (\mathrm{ARG}_0)!$

`FIB` Compute the Fibonacci function *recursively*: $R_0 \leftarrow \mathrm{Fib}(\mathrm{ARG}_0)$

`IS_EVEN` Sets `R0` to 1 if its argument is an even number, 0 otherwise

`IS_NEGATIVE` Sets `R0` to 1 if its argument is a negative number, 0 otherwise

`IS_ODD` Sets `R0` to 1 if its argument is an odd number, 0 otherwise

`IS_POSITIVE` Sets `R0` to 1 if its argument is a positive number, 0 otherwise

`IS_ZERO` Sets `R0` to 1 if its argument is equal to 0. Sets `R0` to 0 otherwise

`POWER` Compute power function: $R_0 \leftarrow \mathrm{ARG}_0{}^{\mathrm{ARG}_1}$.

`SIGNUM` Computes the signum function: `R0` gets 1, 0, or -1, depending on the sign of its argument

4.3 The char library: `char.lib`

`CHAR_IN_RANGE` Return 1 to `R0`, depending whether a char (`ARG[0]`) lies between two other chars (`ARG[1]` and `ARG[2]`): $R_0 \leftarrow \mathrm{ARG}_1 \leq \mathrm{ARG}_0 \leq \mathrm{ARG}_2$

`CHAR_TO_DIGIT` Takes a char in the range ='0' ... '9'=, and places its *numerical* value in `R0`.

```
CHAR_TO_LC R0 <- to_lc(ARG[0])
```

```
CHAR_TO_UC R0 <- to_uc(ARG[0])
```

`DIGIT_TO_CHAR` Takes an integer argument in the range $0, \ldots, 9$, and places the corresponding ASCII value in `R0`.

`IS_CHAR_ALPHABETIC` $R_0 \leftarrow (\texttt{'a'} \leq \mathrm{ARG}_0 \leq \texttt{'z'}) \vee (\texttt{'A'} \leq \mathrm{ARG}_0 \leq \texttt{'Z'})$

`IS_CHAR_LC` $R_0 \leftarrow \texttt{'a'} \leq \mathrm{ARG}_0 \leq \texttt{'z'}$

`IS_CHAR_UC` $R_0 \leftarrow \texttt{'A'} \leq \mathrm{ARG}_0 \leq \texttt{'Z'}$

`IS_CHAR_WHITE_SPACE` Returns 1 if argument is a whitespace char, 0 otherwise. A whitespace is any character that is less than or equal to the space character in the ASCII character set.

4.4 The string library: `string.lib`

`LEFT_STRING` Similar to the `LEFT$` procedure in BASIC: Returns the left substring, given a destination, source, and length. Upon return, register `R0` points to the destination.

`MID_STRING` Similar to the `MID$` procedure in BASIC: Returns the middle substring, given a destination, source, starting position, and length. Upon return, register `R0` points to the destination.

`NUMBER_TO_STRING` Takes a pointer to a dest string and an integer, and writes in the destination the string representation of the number.

`RIGHT_STRING` Similar to the `RIGHT$` procedure in BASIC: Returns the right substring, given a destination, source, and length. Upon return, register `R0` points to the destination.

`STRCAT` Similar to the `strcat` procedure in C: takes a destination, and source, and appends the source to the destination. Upon return, register `R0` points to the destination.

`STRCPY` Similar to the `strcpy` procedure in C: takes a destination, and source, and copies the source to the destination. Upon return, register `R0` points to the destination.

`STRLEN` Takes a pointer to a null-terminated string, and returns its length.

`STRING_TO_LC` Converts the argument string to lowercase. Upon return, register `R0` points to the destination.

`STRING_TO_UC` Converts the argument string to uppercase. Upon return, register `R0` points to the destination.

`STRING_REVERSE` Takes a pointer to a null-terminated string, and reverses it in place.

`STRING_TO_NUMBER` Converts a source string to a number. Similar to atoi in C.

4.5 The system library: `system.lib`

`MALLOC` Moves to R_0 a pointer to a block of as many words as ARG_0. The implementation of `MALLOC` is very primitive, and currently provides no corresponding subroutine `FREE`. This shall be remedied in later releases of the package.

4.6 The Scheme library: `scheme.lib`

The Scheme library is orthogonal to the other libraries. It is concerned with the use of this architecture for the undergraduate *Compiler Construction* course taught at the computer science department of Ben-Gurion University. The library contains assembly routines for creating and managing Scheme data objects: Integers, Booleans, chars, strings, vectors, pairs, etc. If you are not a part of the course and are uninterested in compiling Scheme expressions to machine language, then you may safely ignore this section.

The Scheme library currently makes use of the `MALLOC` procedure. This shall change in later releases of the package, as I complete a full *stop-and-copy* garbage collector, in assembly language. The library shall, at that time, be modified so that the memory it allocates shall be managed automatically by the garbage collector.

`IS_SOB_BOOL` Places 1 in `R0` if its argument is a Scheme Boolean object, or 0 otherwise.

`IS_SOB_CHAR` Places 1 in `R0` if its argument is a Scheme character object, or 0 otherwise.

`IS_SOB_CLOSURE` Places 1 in `R0` if its argument is a Scheme closure object, or 0 otherwise.

`IS_SOB_INTEGER` Places 1 in `R0` if its argument is a Scheme integer object, or 0 otherwise.

`IS_SOB_NIL` Places 1 in `R0` if its argument is a Scheme empty list, or 0 otherwise.

`IS_SOB_PAIR` Places 1 in `R0` if its argument is a Scheme pair, or 0 otherwise.

`IS_SOB_STRING` Places 1 in `R0` if its argument is a Scheme string, or 0 otherwise.

`IS_SOB_SYMBOL` Places 1 in `R0` if its argument is a Scheme symbol, or 0 otherwise.

`IS_SOB_VECTOR` Places 1 in `R0` if its argument is a Scheme vector, or 0 otherwise.

`IS_SOB_VOID` Places 1 in `R0` if its argument is a Scheme void object, or 0 otherwise.

`MAKE_SOB_BOOL` Takes 0 or 1 as an argument, and places in `R0` the corresponding Boolean Scheme object.

`MAKE_SOB_CHAR` Takes an integer between 0 and 255, and places in `R0` the corresponding Scheme character object for the ASCII char.

`MAKE_SOB_CLOSURE` Takes a pointer to an environment, and a pointer to code, and places in `R0` the corresponding Scheme closure object.

`MAKE_SOB_INTEGER` Takes an integer, and places in `R0` the corresponding Scheme integer object.

`MAKE_SOB_NIL` Places in `R0` the Scheme empty list object.

`MAKE_SOB_PAIR` Takes two Scheme objects for arguments, corresponding to the *car* & *cdr*, and places in `R0` the Scheme pair object.

`MAKE_SOB_STRING`

`MAKE_SOB_SYMBOL`

`MAKE_SOB_VECTOR`

`MAKE_SOB_VOID`

`WRITE_SOB`

`WRITE_SOB_BOOL`

`WRITE_SOB_CHAR`

`WRITE_SOB_CLOSURE`

`WRITE_SOB_INTEGER`

`WRITE_SOB_NIL`

`WRITE_SOB_PAIR`

`WRITE_SOB_STRING`

`WRITE_SOB_SYMBOL`

`WRITE_SOB_VECTOR`

`WRITE_SOB_VOID`