6.823 Computer System Architecture
EDSACjr

http://csg.csail.mit.edu/6.823/

The first computer architects did not know exactly what was needed in their machines. They did know, however, that parts were expensive and unreliable. Thus, these pioneers developed architectures that minimized hardware while attempting to provide sufficient functionality to programmers. One of the first electronic computers, EDSAC, had a single accumulator and only absolute addressing of memory. Since one could reference memory only by an address listed explicitly within a program, self-modifying code was essential.

We now know that register based addressing and indirection make assembly level programming and compilation a lot easier. This lesson was learned, however, after programmers spent nearly five years programming absolute addressing machines. Here we introduce an instruction set that gives the functionality of EDSAC without some of the more obscure opcodes. This instruction set, named the EDSACjr, is described in Table H1-1. In the notation used in the table below, M[*x*] stands for the contents of the memory location addressed by *x*. Accum refers to the accumulator. ← signifies that data is transferred (copied) from the location to the right of the ← to the location on the left. The immediate variable *n* is an address or a literal depending on the context. The EDSACjr architecture allows programmers to put constants at any point in the memory when a program is loaded.

Opcode	**Description**	**Bit Representation**
ADD *n*	Accum ← Accum + M[*n*]	00001 *n*
SUB *n*	Accum ← Accum - M[*n*]	10000 *n*
STORE *n*	M[n] ← Accum	00010 *n*
CLEAR	Accum ← 0	00011 00000000000
OR *n*	Accum ← Accum | M[*n*]	00000 *n*
AND *n*	Accum ← Accum & M[*n*]	00100 *n*
SHIFTR *n*	Accum ← Accum shiftr *n*	00101 *n*
SHIFTL *n*	Accum ← Accum shiftl *n*	00110 *n*
BGE *n*	If Accum ≥ 0 then PC ← *n*	00111 *n*
BLT *n*	If Accum < 0 then PC ← *n*	01000 *n*
END	Halt machine	01010 00000000000

Table H1-1: The EDSACjr instruction set

The shifts are arithmetic shifts. All words are 16 bits long. As in EDSAC, instructions are encoded as integers. The first 5 bits are the opcode and the last 11 bits form the immediate field (an 11-bit immediate address addresses up to 2048 words (16-bit) of memory -- twice that of the real EDSAC). Integers are represented in 16 bits, the most significant bit being a sign bit.

Using Macros for EDSACjr

What are Macros?

Macros are assembler directives that allow the programmer to define short names for a sequence of instructions. Once defined, a macro can be used within the code in place of the sequence of instructions, thus saving the programmer time and effort, as well as making the program easier to read. At assembly time, macros are *expanded* according to their definition. That is, each occurrence of a macro is replaced by its corresponding instruction sequence. As described below, macros can have arguments, as well as both global and local labels.

A Simple Example

If you have already tried writing code with the EDSACjr instruction set, you have probably noticed that there is no LOAD instruction for putting the value at a memory location into the accumulator. To do a LOAD, you need to first CLEAR the accumulator, and then ADD the contents of the desired memory location to it. If you need to do a lot of LOADs, it can be quite cumbersome to always have to type this CLEAR-ADD sequence. To make it easier, we can define the following macro for LOAD:

```
.macro LOAD(n)
     CLEAR
     ADD n
.end macro
```

Then, whenever the line LOAD(n) occurs in the code, it will be replaced by the two instructions, CLEAR and ADD as defined. For example:

```
LOAD A          → expands to →      CLEAR
SUB B                               ADD A
                                    SUB B
```

Note how the argument of the macro is used during expansion. It is also possible to use multiple arguments separated by commas.

It is very important to note that a macro is *not* the same as a subroutine or function. A function call involves jumping to and returning from a *single* piece of subroutine code located somewhere in memory, while a macro "call" involves in-place substitution of the macro code. That is, if there are multiple occurrences of a macro, then the macro code is duplicated multiple times, once

at each occurrence in the code. This duplication does not happen with function calls. Because of this difference, using macros usually results in more actual code than using a function call. However, it also usually results in slightly faster code, since macros do not have the calling-convention overhead needed by function calls.[1]

Global Labels

Since macros work by simple expansion, you can refer to labels outside the macro, and these labels will be used verbatim as long as the name of the label does not conflict with any of the macro's arguments or with labels defined within the macro. For example, suppose we define the following macro:

```
.macro STOPGE
     BGE stop
.end macro
```

We may then use this macro as follows:

```
        STOPGE        → expands to →          BGE stop
        SUB B                                 SUB B
stop:   ADD A                           stop: ADD A
```

As shown, global labels are useful for accessing commonly used locations in memory.

Local Labels

You can also define and use labels *within* a macro. During expansion, such local labels will be replaced by a unique label for each instance of the macro. For example, consider the following interesting (silly?) macro:

```
.macro HANGGE
here:
     BGE here  ; if accum >= 0, then loop forever
.end macro
```

When this macro is used, the local labels are expanded as follows:

```
HANGGE        → expands to →      here_1:  BGE here_1
SUB B                                      SUB B
HANGGE                            here_2:  BGE here_2
```

[1] Another interesting difference between macros and function calls is that you cannot write recursive macros. Think of what will happen if you do.

Note that the local label `here` is converted to a unique label during expansion so that multiple instances of a macro do not interfere with each other.

It is also possible to place a label at the end of a macro definition, even if there is no instruction at that position. Such a label would point to the instruction immediately following the macro call in the main code. For example, we can define a conditional ADD macro as follows:

```
.macro ADDGE n      ; this macro only adds if accum >= 0
     BLT donothing  ; if accum < 0,
                    ; then just go to instruction after
                    ; macro
     ADD n          ; else accum <- accum + M[n]
donothing:
.end macro
```

It can be used as follows:

```
ADDGE A      → expands to →                BLT donothing_3
SUB B                                      ADD A
                           donothing_3:    SUB B
```