Machine Considerations for Assemblers

Assemblers are crucial tools in systems programming that convert assembly language code into machine code executable by a computer's CPU. When designing or working with assemblers, several machine considerations must be taken into account to ensure the correct and efficient translation of assembly language to machine code.

Machine Considerations for Assembler Design

1. Instruction Set Architecture (ISA)

An Instruction Set Architecture (ISA) is part of the abstract model of a computer that defines how the CPU is controlled by the software. The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done.

The ISA provides the only way through which a user is able to interact with the hardware. It can be viewed as a programmer’s manual because it’s the portion of the machine that’s visible to the assembly language programmer, the compiler writer, and the application programmer.

The ISA defines the supported data types, the registers, how the hardware manages main memory, key features (such as virtual memory), which instructions a microprocessor can execute, and the input/output model of multiple ISA implementations. The ISA can be extended by adding instructions or other capabilities, or by adding support for larger addresses and data values.

Understanding what the instruction set can do and how the compiler makes use of those instructions can help developers write more efficient code. It can also help them understand the output of the compiler which can be useful for debugging. Arm is opening its instruction set architecture for Cortex M cores. By allowing licensees to build their own custom instructions, developers are able to accelerate specialized workloads. The Arm ISA family allows developers to write software and firmware that

conforms to the Arm specifications, secure in the knowledge that any Arm-based processor will execute it in the same way.

The following are some of the considerations under this category:

1.1 Instruction Formats

There are many instructions in our assembly language, but they can be split into just three categories, by the number and types of operands.

1.1.1 *Register format* instructions have one destination and two sources, all of which are registers e.g ADD R1, R2, R3

1.1.2 *Immediate format* instructions also have one destination register, but the sources include one register and one constant value e.g SUB R1, R2, #2

1.1.3 *Jump and branch format* instructions always need a target address, and there may be a source register as well e.g

```
JMP LABEL1
BZ R2, LABEL2
```

1.2 *Addressing Modes*

The addressing modes help us specify the way in which an operand's effective address is represented in any given instruction. Some addressing modes allow referring to a large range of areas efficiently, like some linear array of addresses along with a list of addresses. The addressing modes describe an efficient and flexible way to define complex effective addresses.

The programs are generally written in high-level languages, as it's a convenient way in which one can define the variables along with the operations that a programmer performs on the variables. This program is later compiled so as to generate the actual machine code. A machine code includes low-level instructions.

A set of low-level instructions has operands and opcodes. An addressing mode has no relation with the opcode part. It basically focuses on presenting the address of the operand in the instructions.

The addressing modes refer to how someone can address any given memory location. Below are some of the common types of addressing modes

1.2.1 Implied Mode - the operands are implicitly specified in the definition of instruction
Immediate Mode - the operand is specified in the instruction itself
1.2.2 Register Mode - the operands exist in those registers that reside within a CPU
1.2.3 Register Indirect Mode - the instruction available to us defines that particular register in the CPU whose contents provides the operand's address in the memory. In simpler words, any selected register would include the address of an operand instead of the operand itself
1.2.4 Direct Address Mode - In the direct address mode, the address part of the instruction is equal to the effective address. The operand would reside in memory, and the address here is given directly by the instruction's address field. The address field would specify the actual branch address in a branch-type instruction.
1.2.5 Indirect Address Mode - In an indirect address mode, the address field of an available instruction gives that address in which the effective address gets stored in memory. The control fetches the instruction available in the memory and then uses its address part in order to (again) access memory to read its effective address.
1.2.6 Indexed Addressing Mode - In the indexed addressing mode, the content of a given index register gets added to an instruction's address part so as to obtain the effective address. Here, the index register refers to a special CPU register that consists of an index value. An instruction's address field defines the beginning address of any data array present in memory

1.3 Instruction Types

Machine instructions generally fall into three categories: data movement, arithmetic/logic, and control-flow. In this section, we will look at important examples of x86 instructions from each category

1.3.1 Data Movement Instructions include mov (move), push (push on stack), pop (pop from stack), lea (load effective address)

1.3.2 Arithmetic and Logic Instructions include add, sub, inc/dec, imul, idiv, and/or/xor, not, neg, shl/shr

1.3.3 Control Flow Instructions include jmp, jcondition, cmp, call, ret

2. Registers

Registers are small, fast storage locations within the CPU used to hold temporary data and instructions. Considerations include:

Number and Types of Registers: General-purpose registers, segment registers, special-purpose registers.

Register Usage Conventions: Which registers are used for specific purposes (e.g., accumulator, index registers, stack pointer).

Example:

In ARM architecture, there are 16 general-purpose registers (R0-R15), with specific roles for some like R13 (stack pointer), R14 (link register), and R15 (program counter).

3. Memory Architecture

The memory architecture affects how instructions and data are stored and accessed.

3.1 Endianness

Endianness is one of the challenges that comes with handling data diversity in assembly. Endianness refers to the byte ordering of multi-byte data types. Some processors store the most significant byte first (big-endian), while others store the least significant byte first (little-endian). When working with multi-byte data types, the

programmer needs to be aware of the endianness of the processor to ensure that the data is stored and retrieved correctly.

3.2 Memory Segmentation:

Segmentation is the process in which the main memory of the computer is logically divided into different segments and each segment has its own base address. It is basically used to enhance the speed of execution of the computer system, so that the processor is able to fetch and execute the data from the memory easily and fast.

Need for Segmentation –

The Bus Interface Unit (BIU) of 8086 microprocessor contains four 16 bit special purpose registers (mentioned below) called as Segment Registers.

Code segment register (CS): is used for addressing memory location in the code segment of the memory, where the executable program is stored.

Data segment register (DS): points to the data segment of the memory where the data is stored.

Extra Segment Register (ES): also refers to a segment in the memory which is another data segment in the memory.

Stack Segment Register (SS): is used for addressing stack segment of the memory. The stack segment is that segment of memory which is used to store stack data.

3.3 Alignment Requirements

Memory alignment is one of the challenges of handling data diversity in assembly programming. Different data types have different memory alignment requirements, which can affect the performance of the program. For instance, some processors require that data be aligned on a 4-byte boundary, while others require an 8-byte boundary. Failing to align the data correctly can result in performance penalties and even program crashes.

4. **Instruction Encoding**

Instruction encoding defines how assembly language instructions are translated into binary machine code. Considerations include:

Opcode Mapping: Mapping of assembly mnemonics to their corresponding opcodes.

Operand Encoding: Encoding of different operand types and addressing modes.

Instruction Length: Fixed-length vs. variable-length instructions.

Example:

In MIPS architecture, instructions are fixed-length (32 bits) with a specific format for R-type, I-type, and J-type instructions.

5. Symbol Management

Symbol management involves handling labels, variables, and constants used in assembly code. Considerations include:

Symbol Table: A data structure used to keep track of the symbols and their addresses or values.

Label Resolution: Mapping labels to their corresponding addresses in memory.

Forward References: Handling references to labels that are defined later in the code.

Example

In an assembly program, a label used in a jump instruction must be resolved to its memory address when the assembler processes the label definition.

6. Macro Processing

Macro processing is an essential aspect of assembler design, playing a critical role in enhancing programming efficiency and readability. Understanding why macro processing is a machine consideration requires delving into the nature of assemblers, the benefits of macro processing, and the machine-level implications of using macros.

Macro processing involves handling macros, which are sequences of instructions that can be reused. Considerations include:

6.1 Macro Definition and Expansion

Support for the definition of macros and their correct expansion in the source code. This involves parsing macro definitions, storing them, and replacing macro invocations with their corresponding code. Defining macros and expanding them during assembly.

6.2 Parameter Handling: Passing parameters to macros and substituting them appropriately.

Example

```
%macro PRINT 1
   mov edx, %1
   mov ecx, msg
   mov ebx, 1
   mov eax, 4
   int 0x80
%endmacro
```

7. Optimization

Optimization involves improving the efficiency of the generated machine code. Considerations include:

7.1 Instruction Scheduling: *{Arranging instructions to avoid pipeline stalls and improve execution efficiency}.*

Instruction scheduling is a compile-time activity that tries to improve the quality of the code that the compiler produces. Like most optimizations, the scheduler runs at compile time. It analyzes the code and reorders the operations before they are emitted by the compiler. The benefits of that rearrangement accrue at runtime.

7.2 Peephole Optimization: {*Small-scale optimizations that simplify or replace sequences of instructions}.*

Most of the compilers produce good code through careful instruction selection and register allocation. A few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying optimizing transformations to the target program. This naive process of statement-by-statement code generation often produce redundant instructions that can be optimize to save time and space requirement of target program.

A simple but effective technique for locally improving the target code is peephole optimization, which examines a short sequence of target instructions in a window (peephole) and replaces the instructions by a faster and/or shorter sequence whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the IR. The peephole is a small, sliding window on a program.

7.3 Dead Code Elimination: Removing instructions that do not affect the program's outcome.

Suppose ***c*** is dead, that is, never subsequently used, at the point where the statement **c**:= a+b appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

```
if(a==b) {
    b=c ;
    return b ;
   c = a + b ;
}
```

8. Debugging and Error Handling

Assemblers need mechanisms for debugging and handling errors in assembly code. Considerations include:

Error Reporting: Providing meaningful error messages and diagnostics for syntax errors, undefined symbols, etc.

Debugging Support: Generating debug information to help in source-level debugging.

Example

An assembler might generate a symbol table with source line information to assist debuggers in mapping machine instructions back to the source code.