CS311 Lecture: The Architecture of a Simple Computer

July 30, 2003

Objectives:

1. To introduce the MARIE architecture developed in Null ch. 4
2. To introduce writing programs in assembly language

Materials:

1. Transparency of Figure 4.8 in Null
2. Transparency of Figure 4.10 in Null
3. Transparency of Table 4.7 in Null
4. Slide to project of first example program
5. Slide to project + handout of second example program in C++
6. Slide to project of machine language translation of this program
7. Slide to project + handout of MARIE assembly listing of this program

I. Introduction

A. The text developed an example of a very simple computer architecture called MARIE - "A Machine Architecture that is Really Intuitive and Easy".

1. Obviously, this machine is hypothetical - no one would actually build a computer system based on this architecture. There is, however, a software simulator for it that we will be using for demonstrations, and which is available on the eMacs downstairs.

2. However, it does exhibit similarities to an actual computer architecture that was manufactured commercially in large quantities (estimated in excess of 300,000) starting in the late 60's and continuing as late as 1990 - the DEC PDP-8.

   (The PDP-8 has a warm spot in my heart, because it was the kind of computer we had when I first began teaching CS at Gordon in 1978!)

B. We will use MARIE to provide a very gentle introduction to assembly-language programming, but we will use a machine using an actual commercial architecture for most of our programming labs. We will come back to MARIE when we get to computer implementation, and will show how this machine could be implemented using basic gates. (At this point you will be glad we're using the MARIE architecture, not the Pentium!)

II. Overview of MARIE

A. Any computer system is composed of three major subsystems:

1. The CPU

2. The Memory

3. The IO System

   These subsystems are interconnected by some sort of bus system. Section 4.1 in the book discusses the system used by Marie, but we will not discuss this now. We will come back to it later in the course.

   TRANSPARENCY: Book Figure 4.8

B. The MARIE IO System is extremely simple - a keyboard and a display, This is one place where the architecture is vastly simplified for pedagogical reasons.

1. In the diagram in the book, the keyboard and display are connected to InREG and OutReg, respectively. Anything typed on the keyboard is read through InReg, and anything placed in OutREG is shown on the display.

2. In a real computer, input and output would typically be in the form of ASCII codes for characters

   a) To display a number, one would output, in succession, the ASCII codes for each of the digits. (So the number 42 would be output as 0x34 0x32.)

   b) To read a number, one would input, in succession, the ASCII codes for each of its digits - stopping when a non-digit ASCII code is read.

3. In the MARIE simulator, it is possible to request that a series of digits typed at the keyboard be converted to binary and placed in InREG, and that a binary number placed in OutReg be displayed as a series of digits on the display. The hardware that would be needed to implement theese capabilities would actually reside in the keyboard and the display, and would be quite complex - but since it's a hypothetical system, that's not an issue.

4. A more conventional system that worked with ASCII codes for individual characters would use exactly the same architecture - but would require much more complex software to convert between binary integers and ASCII codes for decimal characters. (In fact, the simulator makes that option available, too).

C. The MARIE memory system is also quite simple. The memory consists of 4096 words of 16 bits each - numbered 0x000 through 0xfff.

1. It is more common to find actual computer systems today having memory systems that are byte-addressable - so each 8 bit byte has its own address. (If MARIE's memory were byte addressable, the addresses would range from 0x0000 through 0x1fff, and each 16 bit word would occupy two successive bytes.) But commercial computer systems have been built using various addressable units bigger than one byte - including 12 bits, 16 bits, 36 bits, and 60 bits.

2. MARIE's memory could actually be implemented using 2 memory chips - if one could find chips small enough! (Two chips would be needed because it is rare to find a single memory chip that supports access to more than 8 bits at a time due to pinout considerations, so each 16 bit word would come as 8 bits from each of two chips.)

3. A word in memory can be either a data item (a 16-bit integer) or an instruction. From a hardware standpoint, there is no distinction - the meaning of the word is determined from the context in which it is used. (In fact, it is possible for the same word to be, at different times, interpreted as a data item or an instruction. This is also true of "real" computers, and can lead to really strange program behavior when a program (typically inadvertently) modifies itself!)

D. The MARIE CPU consists of an Arithmetic Logic Unit (ALU), a Control Unit, and an interface to memory (MAR and MBR).

1. The ALU and Control Unit each contain a single register.

a) The ALU contains a 16-bit accumulator (AC). This is one of the two operands for all arithmetic operations, and gets the result of the operation. (A role similar to the display on a calculator.)

b) The Control unit contains a 12-bit program counter (PC), which always holds the memory address of the next instruction to be executed.

(The PC is 12 bits wide because memory addresses require 12 bits. If MARIE had a 64K memory, the PC would be 16 bits, etc.)

2. The Control unit repeatedly executes the following process, which is called the Fetch-Decode-Execute cycle:

   a) Fetch a word from memory from the address specified by the PC; then increment the PC.

   b) Decode the word just fetched as an instruction.

   c) Execute the instruction that was decoded.

   d) This process is executed repeatedly (millions of times per second on a modern CPU) until an instruction is executed that halts the CPU. (On modern computers, such an instruction is only executed at the very end of the shutdown process.)

3. During decoding, the word that was fetched from memory is treated as composed of two parts: a 4 bit opcode that specifies what the do, and a 12 bit operand address that specifies the address in memory of an operand of the instruction. (This is ignored for instructions like halt that don't require an operand.)

   TRANSPARENCY - Figure 4.10

4. With a 4 bit opcode, MARIE could have up to 16 different instructions, since $2^4 = 16$. In fact, only 13 of these are used - leaving room, in principle, for 3 more instructions to be added to the instruction set.

   TRANSPARENCY - Table 4.7.

   (In practice, on a machine with small opcodes like this, all the possible values would be used, and various tricks would be used to encode additional instructions - e.g. using bits in the operand field to encode additional operations when the operand field is not needed for a memory address. MARIE actually does this in the case of the Skipcond instruction - opcode 1000 (binary))

E. The text uses a notation called Register-Transfer-Language (RTL) to define each instruction. This is an example of what is called operational semantics - the meaning of an instruction is specified by describing what it does in a language that can be interpreted. That is, if you built a machine that executed the RTL given for the various instructions, you would have built an implementation of the MARIE architecture.

We won't discuss RTL now - we'll come back to it when we discuss CPU implementation, since the RTL given in the book actually assumes some things about the implementation (organization) which could be done differently while still producing the same behavior (architecture).

III. Two Example MARIE programs

A. Our first example lets the user type two numbers, adds them together, and then prints their sum.

1. We'll let the program start at address 0 in memory. Since it is 6 instructions long, we'll use the first unused memory location (6) to store one of the two numbers typed while we input the next. We give the program in both hexadecimal machine code and symbolic form:

   TRANSPARENCY

Address	Contents (hex)	(symbolic)
0	5000	`Input`
1	2006	`Store 6`
2	5000	`Input`
3	3006	`Add 6`
4	6000	`Output`
5	7000	`Halt`

2. DEMO - Be sure to set Input and Output to Decimal; set Stepping off and delay to maximum.

B. We now consider a more complicated example of a MARIE program. The example is deliberately contrived to make use of all the instructions in the MARIE architecture.

1. It is, in fact, precisely the program that might result from compiling the following C++ program, with two assumptions:

   a) Simple IO operations are used for operations on `cin` and `cout`. (Actual compilers would use much more complex library routines for this!)

   b) The `exit()` function is implemented by halting the CPU. (On an actual computer, control would return to the operating system at this point.)

TRANSPARENCY + HANDOUT

2. The following MARIE program translates the above.

a) Assumptions:

(1) The program starts at memory location 100h and runs through 124h.

(2) The constants needed by the program follow immediately after the program code in locations 125h..127h

(a) 125: the address of the array - 140
(b) 126: -1 (ffff)
(c) 127 (1).

(3) The scalar variables used by the program are placed in memory immediately after the constants

(a) 128 range
(b) 129 ind
(c) 12a sel
(d) 12b - a temporary needed when calculating the address of the selected array element x[ind]

(4) The array x[] is placed in memory locations 140h..144h

TRANSPARENCY - Just machine language

b) It should be clear that writing a program - even one as small as this - in machine language would not be a pleasant, nor would debugging or maintaining it be easy. For this reason, one practically never writes a program in machine language. Instead, one writes in an assembly language, which is (for the most part) line-by-line equivalent to machine language. (E.g. each line of code written in assembly language translates into one machine language instruction.)

c) Lets now look at a side-by-side comparison showing both the assembly-language code and the machine language code. The handout we will look at is the "listing" produced by the MARIE assembler, and includes both the source code and the machine language code it generated.

HANDOUT, TRANSPARENCY

Note: An "industrial strength" assembler would not require knowledge as to the placement of the variables in memory, so in several places symbolic values would be used instead of hexadecimal constants, as noted in the comments. Alas, the MARIE simulator's assembler does not handle this.

3. Some things to note about the program itself. (We'll talk about assembly language distinctives in a moment)

   a) In the C++ version of the program, the two subroutines read() and test() are defined first, then main() - which is necessary so that the routines are defined before they are called. In the MARIE version, main comes first because the simulator always starts execution with the physically-first instruction.

   b) Each subroutine (read() and test()) begins with an unused word of memory (here initialized to 0, but it could be anything) followed by the actual code for the subroutine. The reason for this is that the JnS instruction used by the main program uses the first word of the subroutine to store the return address to go back to when the subroutine completes.

      BEGIN DEMO OF PROGRAM - Delay minimum, single stepping on. STEP ONCE. Note how after call to read() location 103 contains 101 - the address of the next instruction in main() after read() is completed - and PC contains 104 - the address of the first actual instruction of read()

   c) The array x[] is stored in successive locations of memory 140h .. 144h.

      DEMO: Step through entering 5 values 1, 2, 3, 4, 5 - note how each shows up, in turn, at 140, 141 ...

      Note: hardwiring the addresses of the array elements like this is not normal practice, but the alternative would involve using an instruction that is not in the MARIE architecture (but arguably should be) or writing self-modifying code - which is not good!

   d) When we calculate the value of range, we take advantage of the fact that x[4] is in the AC after the last Input/Store sequence, so we just have to subtract off x[0].

DEMO: Step through Store range. Note how calculated range gets stored in 128

e) The final instruction of read() is an indirect jump through 103. It goes to 103 and takes this as the address of the next instruction. (103 is not the address of the next instruction - it is the address that contains the address of the next instruction).

DEMO: Step through JumpI. Note how PC becomes 101.

DEMO: Step into test(). Scroll screen so instructions are visible. Enter 1 as index.

f) In test(), after inputting an index, we have to calculate the address of the location where x[ind] is located. Tnis is done by:

Loading the base address of x[] into the AC (140)
Adding the index (1)

DEMO: Step to point where AC contains calculated address 141 Store this calculated address in the temporary at 12b

g) To access x[i], we clear the AC and then use Add indirect to add x[i] to 0. The AddI instruction adds the item whose address is in 12b - i.e. the item at 141, which is x[1], as desired

DEMO: Step through getting x[1] into AC

h) We compare the selected item to range by subtracting. If the result of subtraction is < 0, the selected item was less than range, etc.

DEMO: Step through computation of difference. Note how negative value is represented in two's - complement in AC

i) The translation of the conditional code is a bit contorted! To see if the selected item is less than the range (i.e. the AC is negative), we use a conditional skip if negative to skip over an instruction that jumps to the next test (check for equality). We get to the instruction to load -1 as a result.

DEMO: Step through SkipCond. Note how PC jumps from 11b to 11d.

DEMO: Step through loading -1 into AC

DEMO: Step through Jump to print. Note how PC goes from 11e to 123

DEMO: Step through outputting value and halting program

j) We won't actually execute this here, but notice how the test for 0 is translated - SkipCond with bits 11-10 = 01 translates into `SkipCond 400` (since 400 hex = 0100 0000 0000)

IV. Some Further Comments About Assembly Language

A. The Assembler built into MARIE is a very minimal assembler. Even so, it has several features common to all assemblers.

1. For the most part, there is a one-to-one correspondence between lines of assembly language code and binary machine language words.

2. The assembler allows the use of symbolic names for machine instructions in place of numeric opcodes - e.g. Input instead of 5, etc.

   (Note: the book examples of assembly language source code use mixed case - e.g. JnS - while the listing uses all upper case - e.g. JNS. The MARIE assembler allows either. Different assemblers have different case conventions.)

3. The assembler allows the use of symbolic labels to refer to variables or program lines that are jump targets - e.g. read, test, range, etc.

   A label must be <u>defined</u> exactly once, by appearing at the start of a program line. It becomes a symbolic name for the address in memory where the code generated by the line goes. (Note symbol table at the end of the listing.)

   (Note: the MARIE assembler requires that label definitions be followed by a comma - but these are deleted from the assembly listing!)

4. The assembler allows symbolic labels to be used as the operands of instructions - or literal absolute values can be used. (Note examples of both in the program.)

5. The assembler recognizes certain <u>directives</u> which are instructions to the assembler. Some result in generation of machine language code, and some do not:

a) org specifies where in memory the assembler begins placing generated code

Note example in program

b) hex, dec, and oct are used to specify constants.

Note examples in program: xaddr, mone, one

c) The same directives can also be used to reserve a location in memory for a variable. In my example, I have used hex 0 for this purpose.

(1) Empty words at start of read, test

(2) int variables at the end of the program

B. Most "industrial strength" assemblers have a few capabilities lacking in MARIE's assembler

1. Special directives for creating blocks of memory for variables (instead of using the constant directives for this purpose). Most assemblers have a directive that would allow setting aside a block of memory bigger than one word - for example, this could have been used to reserve memory for the array x[].

   Typical example: something like `space 5`

2. Most assemblers allow symbols to appear in constant definitions - e.g. it would have been nice to be able to write

   `xaddr, hex x` instead of `xaddr, hex 140`

3. Most assemblers allow simple expressions to appear in operand fields - e.g. it would have been nice to be able to write

   `Store x+1` instead of `Store 141`

4. Most assemblers include directives for segmenting the program into sections - at least separate code and data sections. This would allow the translated code to be placed into a region of memory that can be read but not written (either ROM or protected by the operating system)

a) Useful in embedded systems, which actually often place code and constants into ROM

b) Useful in general-purpose systems to protect the code from unplanned changes. (Code that inadvertently modifies itself can do really wierd things!)

C. Many assemblers include additional features, of which the most useful is some sort of macro facility. Basically, this is a facility that allows a single assembly language instruction to be expanded into several machine language instructions using a template.

1. Some assemblers have this capability built-in for certain common cases. We will see examples of this when we get to using the MIPS assembler

2. Most assemblers allow the programmer to define macros.

   EXAMPLE: The MARIE instruction set has no instruction for negating a number - but this operation can be done by using two instructions.

   ASK: Given a variable x, how would you load - x into the AC?

   Clear the AC, then subtract x

   If the MARIE assembler had a macro facility, we could define a new instruction - say

```
LoadN           variable  // Meaning: Load negated
```

   which the assembler would translate into

```
Clear
Subt      variable
```