Lecture C7: Assembly programming

Response to 'Muddiest Part of the Lecture Cards'

(58 respondents, out of 74students)

1) ***The Assembly commands?*** (10 students)
We can and will review the commands during recitation hours. Please read through the following "Assembly language handout".

2) ***I don't understand how to keep all number systems straight. It would help if we had one page about what they each are & how you convert to and from them?*** (1 student)
You now have one page and more, take a look at the following document "number representation".

3) ***The Boolean operations: NOT, AND, OR, XOR?*** (7 students)
Boolean expressions can be formed much like arithmetic operations, but they contain boolean values (either *True/1* or *False/0*), boolean variables (which can be assigned the values *True/1* or *False/0*) and boolean operations.

We use **truth tables** to define the boolean/logic operations:

The **NOT** operation is sometimes known as the *inverse* operation since it simply reverses the state of all bits. A **0** becomes a **1**, and a **1** becomes a **0**.

A	**NOT** A
0	1
1	0

Example: if A = **NOT** B, then B = **NOT** A

The **AND** operator takes 2 parameters, the result of "A **AND** B" results in a **1** if and only if both operands are **1**, otherwise it results in a **0**.

A	B	A **AND** B
0	0	0
0	1	0

1	0	0
1	1	1

The **OR** operator also takes two operators/parameters, the result of "A **OR** B" results in a **1** if one or both of the operands are **1**, otherwise it results in **0**.

A	B	A **OR** B
0	0	0
0	1	1
1	0	1
1	1	1

The **XOR** operator (the Exclusive Or operator) takes two parameters, results in a **1** if and only if **one** of the operands are **1. "**A **XOR** B" results in a **0** if both operands equal **0**, or if both operands equal **1**.

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

There are a number of different equivalences that are good and helpful to know about, for example: **"DeMorgan's Rules"** stating that:

- **NOT** (A **AND** B) is equivalent to (**NOT** A) **OR** (**NOT** B)
- **NOT**(A **OR** B) is equivalent (**NOT** A) **AND** (**NOT** B)

These equivalences can be verified using the following truth table, where you by comparing column 4 and 7 can verify that **NOT**(A **AND** B) is equivalent to **(NOT** A) **OR (NOT** B).

A	B	A **AND** B	**NOT** (A **AND** B)	**NOT** A	**NOT** B	(**NOT** A) **OR** (**NOT** B)
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1

1	1	1	0	0	0	0

As an exercise, use a truth table to verify that the **"Distribution rules"** shown below are correct:

- (A **AND** B) **OR** C is equivalent to (A **OR** C) **AND** (B **OR** C)
- (A **OR** B) **AND** C is equivalent to (A **AND** C) **OR** (B **AND** C)

Boolean variables are useful in computer programs when we have to make decisions. A program can make a decision by evaluating a *condition*. For example, you want to withdraw $100 from your account, then your program has to make sure that there is at lease 100 dollars in the account, thus allowing you to withdraw $100 depends on whether you have enough money in your account or not. Some examples of conditions are:

- the condition **42 = 17** is FALSE
- the condition **42 = X** depends on the value of the variable X.
- the condition **4 > 2** is TRUE
- the condition **4 < 2**is FALSE
- the condition **(4 > 2) AND (4 < 2)** is FALSE
- the condition **NOT(4 > 2)** is FALSE

4) ***How is the assembly language used to construct a more complex program (such as spider program in Ada)?*** (1 student)
Any high level language such as Ada or C++ get by the compiler transformed into a machine language before it can be executed on a CPU. Any program we write using a high-level language can be written directly using the machine language.

An assembly language is just a "human-readable notation" for the machine language (which is a mere pattern of bits) that a specific computer architecture uses. The machine language is made readable by replacing the raw values with symbols called mnemonics.

5) ***The first PRS question?*** (5 students)

"Given the memory locations FEh and FFh both contain 0 (another way to write this is: M[FEh]=0, M [FFh]=0), what do they contain after the following piece of code executes."

load R1, FEh;The register R1 gets assigned the hexadecimal value FE. (R1=FEh)

load R2, [R1] ;Assign register R2 the memory contents of which register R1 holds the address to. (R2=0 since memory location FEh contains the value 0)

load R3, 110b ;The register R3 gets assigned the binary value 110. (R3=110b=6d=6h)

addi R2, R2, R3 ;The register R2 gets assigned the value of adding R2's and R3's values together (R2=0h +6h=6h)

addi R4, R1,R2 ;The register R4 gets assigned the value of adding R1's and R2's values together (R4=FEh+6h=4h)

store R4,[FFh] ; Put the value of register R4 at memory location FFh. (M[FFh]=4h)

halt ;Stop the execution of the program.

So, the correct answer on the PRS question was alternative 2: memory location FEh contains 0,and memory location FFh contains 04h.

7) ***The second PRS question?*** (2 students)
"What is the result of the following operation? 0110 **xor** 0000"

For more information about the **xor** instruction look at question 3) above.

The answer for 0110 **xor** 0000 is: 0110.

How did we arrive at the answer 0110? We need to perform a bitwise **xor** between the two four bit numbers 0110 and 0000. Start by doing an **xor** operation between the bits at position zero 011**0** and 000**0**, **0 xor 0** results in a **0**. Next perform the **xor** operation between bits at position 1: 01**1**0 **xor** 00**0**0 resulting in a **1**. Next perform the **xor** operation between bits at position two: 0**1**10 **xor** 0**0**00 resulting in a **1**. And finally, perform the **xor** operation between the most significant bits (bits at position 3 in this case): **0**110 **xor** **0**000 resulting in a **0.** You now have the 4 bits resulting from the bitwise operations just performed, write them up on a line, each bit at its bit position, which gives us the result: **0110**.

8) ***Why would you ever want to use the value of a memory address in a calculation?*** (1 student)
For example, if you have a *string* (for example the string "Joeb" as shown in small assembly example below), and want to manipulate the 4th item in that string (in this small example that would be the character 'b'). All information you have is the address for the first character in the string. Lets say that the string "MyName" resides at memory location **20**h in the memory (for this example that means that the ASCII value for 'J' resides in memory position 20h, the ASCII value for 'o' in memory position 21h, the ASCII value for 'e' in position 22h and so on).

; A small example that is going to change the string "Joeb" into being the string "JoeB" by replacing the memory position for character 'b' with the ASCII value of Character 'B' (ASCII for character B = 42h)

jmp 50h ;the PC always starts with value 00, so we have to instantiate the PC with the start address of the instructions

org 20h ; my data area resides at memory position 20h
MyName: **db** "Joeb" ;this is my string

org 50h ;the executable code starts at memory position 50h
load R0, MyName ;load register R0 with the address of or string
load R1, 3 ;add a 3 to the memory location in R0 (R1's value + 3 = the address of character 'b' in the string)
addi R2, R0, R1 ;R2 get the value of 23h = the memory position for character 'b'
load R3, 42h ;R3 gets the ASCII value of character 'B'
store R3, [R2] ;store R3's value on the memory position where 'b' used to be
halt ;stop the execution

9) ***How do you know when to store the address into register or the value of address into register?*** (1 student)
T he interpretation depends on the instruction.

10) ***What are the significance of brackets [ ]?*** (and similar questions) (2 students)
It has to do with: direct or indirect addressing.

For example:

- load R1, 42h ; stores the hexadecimal value 42 into register R1
- load R1, [42h] ; assigns the memory contents at address 42h in memory to register R1.

11) ***How do I interpret 0xfe?*** (1 student)
A hexadecimal number can, using the assembly language/simulator used in class today, be written in 3 ways:

- The number starts with '0x', followed by a sequence of hexadecimal digits ('0' up to '9' and 'A' up to 'F'). For example **0xfe**.
- The number starts with '$', followed by a sequence of hexadecimal digits ('0' up to '9' and 'A' up to 'F'). For example **$fe**.
- The number is a sequence of hexadecimal digits ('0' up to '9' and 'A' up to 'F'), but it may not start with a letter. This sequence is followed by an 'h'. A number can always be made to start with a decimal digit by prefixing the number with a '0', so ABh is written as 0ABh. For example **feh**.

12) ***What is db and org?*** (1 student)
db is short for "data byte". It is used to put/store data directly into memory. A data item can be either a number or a string.

Examples:

- db 1,4,9,16,25,36
- db "Hello world",0

org is short for "origin". It is used to define where in memory code and data should reside.**org** takes one parameter, an address. The address has to be a number. Important to remember is that different fragments of code and data may not overlap.

The following example puts the instruction "**load** R0, 2" at the address 60h in memory (60h = 0x60 = $60)

- **org** 60h
  load R0, 2

13) ***Do we have to always translate into binary when doing additions (such as in the PRS question)?*** (1 student)
No, feel free to do the additions in any number system you feel comfortable using, unless the question/ problem specifically ask you to perform the addition using for example binary.

14) ***Memory allocation?*** (1 student)
Sorry, not sure what this questions means.

15) ***What happens when there is an overflow when adding values .. wouldn't the result be wrong?*** (1 student)
In the general case that is correct, an overflow is a runtime error stating that the generated result is too large to fit into the allotted space, which will generate an exception in Ada. But, it depends on your application if it is wrong or not.

16) ***In Concept Question 1, there was some data beneath the halt command. Was that data part of the program or was it just there to inform us of actual values?*** (1 student)
Not in the concept question, maybe you mean one of the other examples?

jmp Start

org 0x30;
Start:
load R0, 0x10;
load R1, [R0];
load R2, [new_number];
Store R1,[new_number];
Store R2, [R0];
halt;

org 0x20;
new_number : **db** 10d

org 0x10;
old_number : **db** 25d;

The lines after **halt** are the data area, that is where we declare where our constants should reside in memory. The hex number **10d** is placed in memory position **20h**, and the **25d** is placed in memory position **10h**.

17) ***Can you differentiate between: R0, R1, R3; 0xfe; [0xfe]; [R1], from the PRS question?*** (1 student)
R0, R1, R2, .. are the names of the registers in the CPU

0xfe is a hexadecimal number, FE

[0xFE] was used when we wanted to read the data in memory position FEh

[R1] was used when we wanted to read the data of the memory position, which value was kept in register R1

18) ***How does the instruction register for the fetch step work?*** (1 student)
The Instruction Register, IR, keeps the bit pattern describing the instruction currently being executed by the CPU. Using our 'Brookshear format', the first 4 bits in IR is the **opcode**, telling us what operation is performed, the least significant 12 bits show us the operands the operation needs.

19) ***The slide for the fetch slide don't seem to match the slide for the decoding instruction?*** (1 student)
That is a correct observation, it will be fixed for next version of my lecture notes. Thanks.

the bit pattern: 35A7 = **store** R5, [A7h]

the bit pattern: 156C = **load** R5, [6Ch]

20) ***If Store R4, [0xff] means R4:=M[ff], isn't R4:=0? because you are storing [0xff] into R4*** (1 students)
If **0** was stored at memory position FF, the register R4 would get assigned value 0.

21) ***With such limited commands, how does machine language do anything useful?*** (1 student)
Remember the machine language we looked at today was for the extremely limited and small 'Brookshear processor'. The machine language for e.g., an Intel processor has very many more and more

complex machine instructions.

22) ***Why does assigning locations and values have to be so complicated?*** (1 student)
I believe most things/programming languages are complicated if we do not get enough time to learn them and practice using them. Hopefully after some more examples shown in for example recitation it will not seem as complicated any longer.

24) ***Is there somewhere to see what readings we could do for lecture/material covered?*** (1 student)
Yes, reading instructions and material are available on the CP web page, which can be reached from the Unified web page.

25) ***Is it fair to say that in general X represents the register/memory and [X[represents the value?*** (1 student)
Yes it is.

26) ***More on computer architecture?*** (1 student)
Can you be a little bit more specific on what is missing?

27) ***What do you suggest I do if I am extremely lost?*** (1 student)
Talk to me after class, send me an email, try and set up a meeting with me, talk to the TAs, attend recitations and office hours, ask TAs about setting up "tutoring sessions", ...

28) ***What valuable purpose does it have to work with a machine language instead of using the time we invest into learning Ada95 deeper or even learning other useful high-level programming languages such as C++ or Java?*** (and similar comments) (2 students)
Most aerospace systems involve writing software at both the embedded level as well as the application level. Without having knowledge/understanding about the underlying architecture of a computer system, it is very hard to use the computer in the most efficient and safe way. CP is meant to be an introduction course that will allow all the students of the class to have a common knowledge base on the subject. When it comes to aerospace software Ada is a dominant language. In addition, languages such as C++ and Java do not enforce good programming practice.

29) ***"No mud"*** (11 students)
Good :-)