CS 2110 Recitation 11
Intro to Assembly

The TAs :)

February 18th, 2019

1 Recitation Outline

1. Pseudo-Ops and Traps
2. Some Assembly Instructions
3. Layout of an Assembly File (2110 Style)
4. Example File (sum.asm)

2 Pseudo-Ops and Traps

Pseudo-Ops

Pseudo-Ops are directions for the assembler that aren't actually instructions in the ISA.

1. `.orig`: Tells the assembler to put this block of code at the desired address
   Ex: `.orig x3000` will put the block of code at address x3000
2. `.stringz "something"`: Will put a string of characters in memory followed by NULL (which is a single ASCII character with the value 0)
   Ex: .stringz "austin" will put the ASCII code for the letter 'a' in the first memory location, the code for 'u' in the second memory location, and so on until putting 'n' in the next-to-last position and NULL (again, the ASCII code 0) in the last memory location as the terminator
3. `.blkw n`: A pseudo-op that will allocate the next n locations of memory for a specified label.
4. `.fill value`: Pseudo-op that will put that value in that memory location.
5. `.end`: A pseudo-op that denotes the end of an address block. Matches with an .orig

A more visual example:

Instructions
.orig x3000
string_label .stringz "Hey there!"
block_label .blkw 5
fill_label .fill x1234
.end

Address	
x2FFE	Don't know
x2FFF	
x3000	"H"
x3001	"e"
x3002	"y"
x3003	" "
x3004	"t"
x3005	"h"
x3006	"e"
x3007	"r"
x3008	"e"
x3009	"!"
x300A	0 (NULL)
x300B	block_label
x300C	
x300D	
x300E	
x300F	
x3010	x1234
x3011	Don't know either
x3012	
x3013	

Traps

Traps are subroutines built into the LC-3 to help simplify instructions. Some helpful TRAPS include:

1. HALT: Halt is an alias for a TRAP that will stop the LC-3 from running.
2. OUT: Out is an alias for a TRAP that will take whatever character is in R0 and print it to the console
3. PUTS: Puts is an alias for a TRAP that will print a string of characters with the starting address saved in R0 until it reaches a null (0) character
4. GETC: Getc is another TRAP alias that takes in a character input and stores it in R0

Being an alias for a TRAP instruction means that the assembler will convert them to TRAP instructions at assembly time. For example, if you write HALT in your code, the assembler will convert it to the instruction TRAP x25 for you.

Example

```
.orig x3000             ;this is where our code will begin
LEA R0, HW              ;loads the address of our string into R0
PUTS                    ;will print the string whose address is in R0
HALT                    ;stops the program from executing
HW .stringz "Hello. \n" ;stores the word 'Hello' in memory with 'H' located at address x3003,
                        ;'e' will be located at address x3004, etc
.end                    ;denotes the end of the address block
```

3 Notable Assembly Instructions

BR

The BR assembly instruction is extremely helpful for loops or conditionals.

Loops

If you wanted to code a for-loop that took a number and multiplied it by 2, an example java snippet could be:

```
int number = 5;
int result = 0;
for (int i = 0; i < number; i++) {
    result = result + 2;
}
```

Note: Please do not use a for-loop to multiply in java, this code is just to prove a point.
Going directly off of this code, we can formulate the following assembly code:

```
.orig x3000

     AND R0, R0, 0 ;clears out R0
     AND R1, R1, 0 ;clears out R1 - our result
     ADD R0, R0, 5 ;adds 5 to R0 - R0 is our number that we will decrement

LOOP ADD R1, R1, 2
     ADD R0, R0, -1;our iterator
     BRP LOOP      ;checks the conditional code.
                   ;Until our iterator reaches zero, we will keep branching to the label LOOP
     HALT
.end
```

Note: The biggest thing with loops is that one register will hold the iterator and decrements each iteration of the loop. The BRP will only keep branching to the top of the loop code while this iterator remains positive.

Conditionals

What if you wanted to mimic the following java code:

```
    int result;
    if (x < 0) {
        result = 10;
    } else {
        result = 5;
    }
```

In assembly, you would put 'X' in a register and then branch based off of conditional code. With conditionals, remember to branch out of the entire statement at the end!

```
.orig x3000
    AND R0, R0, 0   ; clears out R0 - will be our result register
    AND R1, R1, 0   ; clears out R1
    ADD R1, R1, X   ; adds X to R1, this instruction will set the conditional code (cc)

    BRZP NONNEG     ;if the cc is zero/positive, will jump to the NONNEG label
    ADD R0, R0, 10  ;this code will only be run if we didn't branch - the cc was negative
    BRZNP CONT      ;will allow us to escape the conditional statement and not run the 'else' clause

NONNEG
    ADD R0, R0, 5   ;the cc was zero/pos
CONT
HALT
```

LD

Your basic load operation. This instruction loads data at a certain memory address into the designated register

```
.orig x3000
    LD R0, X    ;loads the value at the memory address X, 2 -> R0
    HALT
X   .fill 2     ;the memory address labeled 'X' contains the value 2
.end
```

LDR

A load operation that takes the memory address from a register, adds it to an offset, retrieves the data at the subsequent offset + address memory address, and loads it into the designated register. Ex:

```
.orig x3000
     LD  R0, HELLO  ;loads the address of the label HELLO into R0
     LDR R1, R0, 1  ;takes the memory address x6000 located in R0 and adds 1 -> x6001
                    ;Goes into memory at x6001 and retrieves the data 'i'
                    ;and places the ASCII value in R1
    HALT
HELLO .fill x6000
.end

.orig x6000
.stringz "Hiya\n"
.end
```

LEA

LEA or load effective address, will load the memory address into a designated register. This is extremely helpful for printing strings, because you can load the address of the starting character into R0 via LEA and then call PUTS

```
.orig x3000
    LEA R0, BESTCS ;will load the address represented by the BESTCS label into R0
    PUTS           ;using the starting address located in R0
                   ;will print a string of characters until reaching the null character
HALT
BESTCS .stringz "CS2110\\n"
.end
```

ST

ST is your basic storing operation that will store a particular value at a certain memory address. Ex:

```
.orig x3000
    AND R0, R0, 0   ;clear R0
    ADD R0, R0, 15  ;puts 15 in R0
    ST  R0, ANSWER  ;this operation stores whatever is in R0 (15)
                    ;into the memory address represented by ANSWER
HALT
.end
ANSWER .blkw 1      ;blocks one word of memory
```

STR

STR is similar to ST, except that the memory address is calculated by adding a base address to a value located in a register.

4 Layout of an Assembly file (2110 Style)

Take a look at the below code. The first .orig and .end corresponds to the instruction part of memory located at x3000, where we put all of our instructions and some local variables. The second .orig and .end correspond to the data part of memory located at x6000 where more data is located.

```
.orig x3000
    LD R1, B
    LD R0, A
HALT
A     .fill 2
ARRAY .fill x6000
.end
.orig x6000
1
2
3
.end
```

The below image shows the memory image of this piece of code. The biggest thing to note is that a Label, such as A or Array in this piece of code, just represents a particular memory address.

	Address	
	x2FFE	Don't know
	x2FFF	
	x3000	LD R1, B
	x3001	LD R0, A
	x3002	HALT
A	x3003	2
ARRAY	x3004	x6000
	x3005	Don't know
	x5FFE	Don't know
	x5FFF	
	x6000	1
	x6001	2
	x6002	3
	x6003	Don't know
	x6004	

5 Example

Sum the values from the memory locations labelled X and Y, then print a message and their sum to the user.
Pseudocode:

```
void sum() {
    int x = 2;
    int y = 3;
    int sum = x + y;
```

```
    const char *msg = "The sum is: ";
    printf(msg);
    printf("\%c", sum + 48); //we need a 48 here because the OUT trap prints the number as ASCII
}
```

Setup:

```
.orig x3000                     ;beginning of our code
    ;code goes here
HALT                            ;ensures that nothing will be executed beyond this point

HELLO   .stringz "The sum is: "
X       .fill 2
Y       .fill 3
TOASCII .fill 48
.end                            ;end of our code
```

First off, we want to load the values of X, Y, and TOASCII into registers. Because we want to just load the values at these memory addresses, we will use the basic LD instruction.

```
.orig x3000                     ;beginning of our code
    LD R1, X                    ; 2 -> R1
    LD R2, Y                    ; 3 -> R2
    LD R3, TOASCII              ; 48 -> R3
    ADD R1, R2, R1              ; R1 = 2 + 3 = 5
    ADD R1, R1, R3              ; R1 = 5 + 48 = 53 (ASCII value of 5)

    ;more code will go here
HALT                            ;ensures that nothing will be executed beyond this point

HELLO   .stringz "The sum is: "
X       .fill 2
Y       .fill 3
TOASCII .fill 48
.end                            ;end of our code
```

Next, we want to print out the string "The sum is: ". The easiest way of doing this is loading the starting address of the string into R0 using LEA then calling PUTS.

```
.orig x3000                     ;beginning of our code
    LD R1, X                    ; 2 -> R1
    LD R2, Y                    ; 3 -> R2
    LD R3, TOASCII              ; 48 -> R3
    ADD R1, R2, R1              ; R1 = 2 + 3 = 5
    ADD R1, R1, R3              ; R1 = 5 + 48 = 53 (ASCII value of 5)
    LEA R0, HELLO               ; Starting address of string located in R0
    PUTS                        ; This traps takes the starting address located in R0
                                ; and will print characters until it reaches a null terminator

    ;more code will go here
HALT                            ;ensures that nothing will be executed beyond this point

HELLO   .stringz "The sum is: "
X       .fill 2
Y       .fill 3
TOASCII .fill 48
.end                            ;end of our code
```

Finally, we can print our sum by transferring it to R0 and calling OUT.

```
.orig x3000                     ;beginning of our code
    LD R1, X                    ; 2 -> R1
    LD R2, Y                    ; 3 -> R2
    LD R3, TOASCII              ; 48 -> R3
    ADD R1, R2, R1              ; R1 = 2 + 3 = 5
    ADD R1, R1, R3              ; R1 = 5 + 48 = 53 (ASCII value of 5)
    LEA R0, HELLO               ; Starting address of string located in R0
    PUTS                        ; This trap takes the starting address located in R0
                                ;and will print characters until it reaches a null terminator
    ADD R0, R1, 0               ; 53 -> R0 (which is the ASCII value of 5)
    OUT                         ;This trap takes the ASCII value located in R0
                                ;and prints it to the screen
HALT                            ;ensures that nothing will be executed beyond this point

HELLO   .stringz "The sum is: "
X       .fill 2
Y       .fill 3
TOASCII .fill 48
.end                            ;end of our code
```