LC-3 Assembly Language and the Assembler

Assembly Language

- Assembling with an Assembler
- Coding is exactly what we have been doing so far with a few additions.
  - Instructions: add, and, not, ld, st, ldr, str, etc …
  - Pseudo-ops: .orig, .fill, .blkw, .stringz, .end
  - Labels
- Much easier for humans.
- Addresses are often calculated automatically.
- Computer does more work in the background.
- ; begins a comment
- # is used to mean immediate

DO NOT CONVERT CODE TO HEX BY HAND!!!

- Last week you created your assembly and then converted it by hand.
- I wanted you to learn hex codes for ALL the operations.
- But now, we are going to use the assembler.
- **THIS WEEK YOU SHOULD USE THE ASSEMBLER INSTEAD!!!**
- Write your code in assembly as described in the first part of these notes and save as an ASM file.
- Load your program into the assembler. You can download it from the resources portion of AsULearn.
- Load the program into the assembler and THE ASSEMBLER WILL CONVERT IT TO HEX.
- The assembler will calculate offsets AND store data for you.

Assembly Code Label

- The first column in an assembly program is used for labels.
- A label is simply a way of telling the assembler to remember a specific memory location.
- Labels can be used as positions for branching and jumping to subroutines (BR and JSR).
- Labels can be used like variable names for loading and storing (LD, LDI, LEA, ST, STI).
- The assembler will calculate the proper offsets for you using the label addresses. This only works for PCOffset instructions.

DO NOT USE OFFSETS!!!

- ***<u>From this point on you should NOT calculate or use any numerical offsets in your code!!!</u>***
- LD, ST, BR, LEA, JSR
- These are WRONG and WILL FAIL!!

```
LD R1, #5
BRN #-7
LEA R0, 5
JSR 100
```

- CORRECT!! USE LABELS!! NO NUMBERS!!

```
LD R1, N48
BRN TOP
LEA R0, PROMPT
JSR GETNUM
```

- SEE NEXT SLIDE FOR DETAILS

A simple example - LABELS

```
        .orig   x3000       ; load the program to memory location x3000
        LD      R0, A       ; load the value stored at A to register 0
        LD      R1, B       ; load the value stored at B to register 1
        NOT     R2, R1      ; Invert R1 and store in R2 for making R2 = -R1
        ADD     R2, R2, #1  ; Add R1 to R2 for making R2 = -R1
        ADD     R2, R2, R0  ; R2 <- R0 – R1
        BRN     L1          ; Jump to L1 if R1 > R0
        OUT                 ; Print R0 to the screen since it was larger
        BRNZP   END         ; Jump to end of program
L1      ADD     R0, R1, #0  ; Move R1 to R0
        OUT                 ; Print what was in R1 since it was larger
END     HALT                ; Halt the program
A       .fill   35          ; Variable A is just this memory location
B       .fill   36          ; Variable B is just this memory location
        .end
```

Pseudo Ops – Assembler Directives

- These are special instructions to the assembler, not part of the ISA.
- .ORIG
  - specify the starting address
- .FILL
  - Initialize a single memory location to some value
- .BLKW - BLocK of Words
  - Reserve a block of memory locations
- .STRINGZ
  - Reserve character memory as a string
  - Will be null terminated
- .END
  - Tells the assembler where the program ends

.FILL and .BLKW

- Save a whole range of memory.
- Affects the next memory location.

```
        .orig x3000
        LEA R6, C
        LD R0, A
        HALT
A       .fill 65
B       .fill 66
C       .BLKW 5
D       .fill 68
        .end
```

3000		EC04
3002		2001
3002		F025
3003	A	0041
3004	B	00426
3005	C	0000
3006		0000
3007		0000
3008		0000
3009		0000
300A	D	0044

.STRINGZ

```
        .ORIG x3000
        LEA R0, HELLO
        PUTS
        HALT

HELLO   .STRINGZ "Hello, World!"

A       .fill 5
```

3000	E002
3001	F022
3002	F025
3003	0048
3004	0065
3005	006C
3006	006C
3007	006F
3008	002C
3009	0020
300A	0057
300B	006F
300C	0072
300D	006C
300E	0064
300F	0021
3010	0000
3011	0005

Traps

- Traps are system subroutines used for IO and halting.
- Use the following instead of the trap in assembly language programs.

  Instead of TRAP x20 use **GETC**

  Instead of TRAP x21 use **OUT**

  Instead of TRAP x22 use **PUTS**

  Instead of TRAP x23 use **IN (Don't use this one at all)**

  Instead of TRAP x24 use **PUTSP (Don't use this one at all)**

  Instead of TRAP x25 use **HALT**
- Using the TRAP x## instead of the word shown above will cost you points on the exam and quizzes.

A simple example - PSUEDO-OPS

```
        .orig   x3000          ; load the program to memory location x3000
        LD      R0, A          ; load the value stored at A to register 0
        LD      R1, B          ; load the value stored at B to register 1
        NOT     R2, R1         ; Invert R1 and store in R2 for making R2 = -R1
        ADD     R2, R2, #1     ; Add R1 to R2 for making R2 = -R1
        ADD     R2, R2, R0     ; R2 <- R0 – R1
        BRN     L1             ; Jump to L1 if R1 > R0
        HALT                   ; Print R0 to the screen since it was larger
        BRNPZ   END            ; Branch to end of program
L1      ADD     R0, R1, #0     ; Move R1 to R0
        OUT                    ; Print what was in R1 since it was larger
END     HALT                   ; Halt the program
A       .fill   35             ; Variable A is just this memory location
B       .fill   36             ; Variable B is just this memory location
        .end
```

A simple example (TRAPS)

```
        .orig   x3000        ; load the program to memory location x3000
        LD      R0, A        ; load the value stored at A to register 0
        LD      R1, B        ; load the value stored at B to register 1
        NOT     R2, R1       ; Invert R1 and store in R2 for making R2 = -R1
        ADD     R2, R2, #1   ; Add R1 to R2 for making R2 = -R1
        ADD     R2, R2, R0   ; R2 <- R0 – R1
        BRN     L1           ; Jump to L1 if R1 > R0
        OUT                  ; Print R0 to the screen since it was larger
        BRNZP   END          ; Jump to end of program
L1      ADD     R0, R1, #0   ; Move R1 to R0
        OUT                  ; Print what was in R1 since it was larger
END     HALT                 ; Halt the program
A       .fill   35           ; Variable A is just this memory location
B       .fill   36           ; Variable B is just this memory location
        .end
```

Loops with Labels

```
        ;with labels
        AND R0, R0, #0
        AND R1, R1, #0
        ADD R0, R0, #6
TOP
        ADD R1, R1, R0
        ADD R0, R0, #-1
        BRP   TOP
        HALT
```

```
;without labels
AND R0, R0, #0
AND R1, R1, #0
ADD R0, R0, #6
ADD R1, R1, R0
ADD R0, R0, #-1
BRP #-3
HALT
```

If – Else with Labels

High Level Language (Like Java or C)		Assembly
		.orig x3000

if (R1 > R2)
 	
{
R1 = R1 – R2 //Block 1
}
else
{
R1 = R2 – R1 //Block 2
}
	If R1 < R2
	or R1 = R2
	skip to BLOCK2
		NOT R3, R2
		ADD R3, R3, #1
		ADD R3, R1, R3
		BRNZ **Block2**

	BLOCK1	NOT R2, R2
		ADD R2, R2, #1
		ADD R1, R2, R1

	skip BLOCK2	BRNZP **IF_END_1**

	BLOCK2	**Block2**
		NOT R1, R1
		ADD R1, R1, #1
		ADD R1, R2, R1

		IF_END_1 HALT .end

Assembly Format

- Labels should be all the way to the left margins.
- Instructions should be tabbed once.
- Pseudo-ops should be tabbed once.
- Any operands after the instruction should be separated by one space.
- If there is a comma, it should immediately FOLLOW the thing it is separating.
- Comments may start on the left margin, be tabbed, or follow the instruction.
- See next page for details.
- Not following these rules can cost points on exams and quizzes.

How to Assemble

- Download the assembler.
- Open the .asm file.
- It will automatically assemble on load or will reassemble when the button is clicked.
- Note the lower section of the assembler shows any errors or the last assemble time and date.
- Make sure to check for errors when assembling.
- You can leave the assembler open and have your editor open at the same time for quick update and assemble.

How to Assemble Example

```
            .orig x3000 ;puts the value 3000 as the first line in the hex file

;Code starts here.
MAIN        ;Main is a label.  No one said you have to use it.
            LEA R6, C ;Note no index, only a label
            LD R0, A  ;Note no index, only a label
            HALT        ;Stop the program

;The assembler will ignore all blank lines and comments.
;The hex file will not be changed.

;Data section
A           .fill 65  ;puts 65 into 3003
B           .fill 66  ;puts 66 into 3004
C           .BLKW 5   ;puts 0 into 3005, 3006, 3007, 3008, 3009
D           .fill 68  ;puts 68 into 300A
;See video on AsULearn

;Last thing in file should always be .end
            .end
```

Assembly Results – The hex file

3000
EC04
2001
F025
0041
0042
0000
0000
0000
0000
0000
0044

Homework 3 & 4

- USE THE DOCUMENTS ON ASULEARN FOR DETAILS ABOUT THESE ASSIGNMENTS

JSR and JSRR

- Jump Save Return (JSR)
- Jump Save Return Register (JSRR)
- BOTH SAVE THE RETURN ADDRESS!
- ***JSR*** and ***JSRR*** should ONLY be used if you plan on running a subroutine and RETURNING from that subroutine.
- A subroutine is like a method or function. You jump away to do some code, then return.
- If you just want to jump somewhere and maybe not return, use ***JMP or BRNZP*** instead of ***JSR*** or ***JSRR***.
- ***JSR*** AND ***JSRR*** SHOULD ALWAYS BE USED IN CONJUNCTION WITH THE ***RET*** instruction.

Subroutines

- You can read about subroutines in chapter 9.
- A subroutine is like a function or method.
- Pass parameters using registers.
- Return values using registers.
- When using a subroutine, ALWAYS save and restore any registers that you use in the subroutine.
  - At the beginning of the subroutine save all registers that you use in the subroutine.
  - At the end of the subroutine, before the RET, restore the registers to their original value.
  - DON'T SAVE OR RESTORE THE RETURN REGISTER.

Passing Arguments to Subroutines

- Two solutions
  - Store arguments in memory (preferred but a bit complicated).
  - Store arguments in registers (easy but has drawbacks).
- For now we will assume Registers have the arguments.
- Most high-level languages store arguments in memory on the stack.

The Register Modification Problem

- To work, MULT probably needs to use multiple registers.
- But what if the code that needs to USE mult is using the same registers?
- What if MULT overwrites an important value in a register that the original code needed?
- The code that needs to use MULT is known as the ***CALLER.***
- The code that is getting called (MULT in this case) is known as the ***CALLEE.***
- To prevent register overwriting either the callee or the caller must save the registers and restore them.

CALLEE vs CALLER

- The caller may not know exactly what registers the callee will be using. Therefore, to be safe, the caller would have to save ALL important registers.
- The callee, however, knows exactly what registers it needs to perform its function. Therefore the callee knows exactly which registers to save and restore.
- In general, the callee saving registers is most commonly used and should be used in this class.
- If you are calling a callee and it specifically modifies a register for returning results, it is up to the caller to save the register BEFORE calling the callee.
- FOLLOW THESE RULES
  - Anytime you write a subroutine, the FIRST THING you should do inside of that subroutine is to save (store) the initial value of any register you will be using.
  - Anytime you write a subroutine, that last thing you should do before returning (RET) is to restore ALL of the registers that you saved.
  - You should NOT save or restore any register that you are using to pass a result back to the caller.
  - Anytime you CALL a subroutine, the CALLER must save any registers that it knows will be overwritten by the subroutine.
  - If you wish to call a subroutine or use a TRAP from inside of a subroutine, the inner subroutine will modify R7 and the outer subroutine will not be able to return properly. ANYTIME you use a subroutine or trap from inside of another subroutine or trap you must save R7 and restore R7 before returning.

How to save Registers

```
MULT
        ;Save registers.
        ;If I am using R0, R1, and R2, I need to save the original
        ;values.  I DON'T need to save R0 because that is where
        ;the result of my MULT will be stored.
        st      R1, SAVE_R1     ;Save R1 so it can be restored
        st      R2, SAVE_R2     ;Save R2 so it can be restored

        ;Do any code here

        ;Restore registers
        ld      R1, SAVE_R1     ;Restore R1 before returning
        ld      R2, SAVE_R2     ;Restore R2 before returning

        ;Only return after restoring the registers.
        ret

SAVE_R1 .fill   0
SAVE_R2 .fill   0
        ;Later we will see how to use a stack to save registers.
```

```
            .orig x3000
            JSR         TEST
            HALT

;Describe TEST
TEST
            ST  R1, T_SR1
            ST  R2, T_SR2
            ST  R7, T_SR7

            ;Code
            OUT
            BRNZP T_END

T_END       LD R1, T_SR1
            LD R2, T_SR2
            LD R7, T_SR7
            RET
T_SR1       .fill       0
T_SR2       .fill       0
T_SR7       .fill       0

;A longer example
```

Save and restore ANY register that gets used in your subroutine.
DON'T save any register that is designated a return register.
Register save labels must be unique to each subroutine.

Rules For Assembly Programs

- Here are some rules I am going to require. These are not necessarily always true for assembly programming, but I want you to follow them for easier grading and problem detecting. The tests will be looking for these.
  - Your program must have **exactly** one **HALT**.
  - The HALT MUST be reached. If your program goes into an infinite loop, it will fail the tests. Run the simulator until HALT is reached.
  - Each JSR or JSRR should have **exactly** one **RET**.
  - Any call to a subroutine must NOT modify any registers other than stated return registers. You can use any register as long as you save and restore it.

2 Pass Assembly

- Pass 1
  - Read the code and determine addresses.
  - Associate labels with addresses.
  - assign5.sym

```
//Symbol Name         Page Address
//----------------    ------------
//     A                      3006
//     B                      3007
//     C                      3008
//     MTOP                   300D
//     MULT                   3009
//     SAVE_R1                3013
//     SAVE_R2                3014
```

Assembly List file

assign5.lst

```
(0000) 3000  0011000000000000 (   2)                        .ORIG x3000
(3000) 2005  0010000000000101 (   3)                        LD    R0 A
(3001) 2205  0010001000000101 (   4)                        LD    R1 B
(3002) 4806  0100100000000110 (   5)                        JSR   MULT
(3003) F021  1111000000100001 (   6)                        TRAP  x21
(3004) 3003  0011000000000011 (   7)                        ST    R0 C
(3005) F025  1111000000100101 (   8)                        TRAP  x25
(3006) 0008  0000000000001000 (   9) A                      .FILL x0008
(3007) 0009  0000000000001001 (  10) B                      .FILL x0009
(3008) 0000  0000000000000000 (  11) C                      .FILL x0000
(3009) 3209  0011001000001001 (  22) MULT                   ST    R1 SAVE_R1
(300A) 3409  0011010000001001 (  23)                        ST    R2 SAVE_R2
(300B) 1420  0001010000100000 (  26)                        ADD   R2 R0 #0
(300C) 5020  0101000000100000 (  27)                        AND   R0 R0 #0
(300D) 1040  0001000001000000 (  33) MTOP                   ADD   R0 R1 R0
(300E) 14BF  0001010010111111 (  34)                        ADD   R2 R2 #-1
(300F) 03FD  0000001111111101 (  35)                        BRP   MTOP
(3010) 2202  0010001000000010 (  38)                        LD    R1 SAVE_R1
(3011) 2402  0010010000000010 (  39)                        LD    R2 SAVE_R2
(3012) C1C0  1100000111000000 (  40)                        RET
(3013) 0000  0000000000000000 (  51) SAVE_R1                .FILL x0000
(3014) 0000  0000000000000000 (  52) SAVE_R2                .FILL x0000
```

Use the following to show an example of editor-assembler-simulator usage

[A longer example]

Concepts from the Simulator

- These commands are from the text based simulator (lc3sim) on student. But these ideas are common to debuggers in many languages and IDEs.
- The graphical based simulators have buttons which do the same things as the following commands.
- Type ***help*** from within ***lc3sim*** get a list of the commands.

0.01

step - Execute 1 program step at a time. Will go into subroutines.

next - Execute 1 program step at a time. Will run subroutines and return without stepping through them. Treats JSR like a single step.

finish - If you are in a subroutine will execute until the subroutine finishes then go back to step mode.

continue - Will run to the end of the program with no more stepping.

printregs - Lists all of your registers and values on the screen.

dump - Lists memory and values on the screen.

Homework 5

- **USE THE ASSIGNMENT DOCUMENT ON ASULEARN FOR DETAILS ON THIS ASSIGNMENT.**

Create GETNUM

```
        .orig x3000
        JSR    GETNUM
        JSR    GETNUM
        HALT

GETNUM
        //Program code here
        RET
```

Boolean AND

```
if (A>B && A>C)
{
        print(A)
{
```

- Booleans are positional. Must PASS all tests.
  - If first test fails jump to end of ***if***.
  - If second test fails skip to end of ***if***.
  - Print A
  - End of ***if***

	Assume R0<-A, R1<-B, R2<-C	Comments

	Not R3, R0
	Add R3, R3, #1
	Add R4, R3, R1
	BRZP ENDIF
		;Compare A > B
		
		;leave -A in R3
		;if A<B || A==B

	Add R4, R3, R2 BRZP ENDIF	;Compare A>C ;if A<C || A==C
	Trap x21	;Print R0(A)
ENDIF	Trap x25	

Multiple ANDs

- if (test1 && test2 && test3 && … && testn) {BLOCK}

```
        if test1 fails jump to END
        if test2 fails jump to END
        if test3 fails jump to END
        if …
        if testn fails jump to END
        //Execute the BLOCK of code here
END     //Move on
```

Boolean OR

```
if (A>B || A>C)
{
        print(A)
{
```

- Booleans are positional. Must pass either test.
  - If first test passes jump to ***Print A.***
  - If second test fails skip to end of ***if***.
  - Print A
  - End of ***if***

	Assume R0<-A, R1<-B, R2<-C	Comments

	Not R3, R0
	Add R3, R3, #1
	Add R4, R3, R1
	BRN PRINTA
		;Compare A > B
		
		;leave -A in R3
		;if A>B

	Add R4, R3, R2 BRZP ENDIF	;Compare A>C ;if A<C || A==C
PRINTA	Trap x21	;Print R0(A)
ENDIF	Trap x25	

Multiple ORs

- if (test1 || test2 || test3 || … || testn) {BLOCK}

```
        if test1 passes jump to CODE
        if test2 passes jump to CODE
        if test3 passes jump to CODE
        if …
        if testn passes jump to CODE
        BRNZP END
CODE
        // Execute BLOCK of code here
END
```

Mixed ANDs / ORs

- if (test1 && test2 || test3) {BLOCK}

```
        if test1 fails branch to T3
        if test2 fails branch to T3
        BRNZP CODE
T3
        if test3 fails jump to ENDIF
CODE
        // Execute BLOCK of code here
ENDIF
```

I will count off points on tests

- Setting something to zero when you don't need to.
- Not formatting correctly
  - All labels begin on the left margin.
  - Instructions and pseud-ops should be tabbed once.
  - Instructions with multiple fields must have those fields separated by one space.
  - If a comma is present it should follow immediately the thing it is separating and be followed by a space.
- Not including a .orig or a halt when I ask for an LC3 program.
- Not including an initial label or a RET when I ask for an LC3 subroutine.
- Loading a positive number with a .fill and then making it negative with code. Just store the negative!
- Using two HALT instructions.
- Using more than one RET in a subroutine.
- Using an offset in an instruction when you should use a label.
- Adding more than twice to get some constant. If it isn't possible to add or subtract twice, use a .fill and LD.
- Reloading constants that should have been loaded once, especially if it occurs inside a loop.
- Using the TRAP instruction instead of the proper assembly word. Use GETC, PUT, OUT, HALT.
- Doing anything in a way that is WAY more complicated than necessary. If your code uses advanced features that we did not cover and looks as if you simply copied and pasted something from the Internet, you will receive a zero for the question. I covered the things you needed for this class. The assignments are written assuming you understood those topics. The programs are relatively simple and straightforward. Adding a bunch of unnecessary code or using extremely complicated code shows me that you don't understand what you are doing, and it will be penalized heavily.

When NOT to Clear a Register

- In the following cases the value in R0 will be overwritten by the operation.
  - DO NOT clear a register before GETC.
  - DO NOT clear a register before LD, LDR, LDI
  - DO NOT clear a register after an add like this:

    ADD R1, R2, #5 (R1 = R2 + 5)

    ADD R1, R2, R3 (R1 = R2 + R3)
- When to clear registers.
  - If you are using the register as an accumulator in a loop. Clear the value before the loopl

When to Clear a Register

- When to clear registers.
  – If you are using the register as an accumulator in a loop. Clear the value before the loop.

```
        AND R1, R1, #0    (Clear R1 because you want to start at 0)
TOP     ADD R1, R1, R2    (This is summing up a bunch of R2s in R1)
        BRN TOP
```

- When using immediate add to load a value as a constant.

```
AND R0, R0, #0
ADD R0, R0, #10(You want R0 to be 10.  For this to work R0 must  start at
zero)
```

Summary of Register Clearing

Any time you are adding a value to itself:

R0 = R0 + R1

Where the same register is on both sides of =, you need to think about clearing.

However this:

R0 = R1 + R2

Simply adds R1 and R2 and overwrites R0 and there is no need to clear R0.