Introduction to Computer Engineering

CS/ECE 252, Spring 2017
Rahul Nayar
Computer Sciences Department
University of Wisconsin – Madison
Chapter 7 & 9.2
Assembly Language and Subroutines
Human-Readable Machine Language

Computers like ones and zeros…

0001110010000110

Humans like symbols…

ADD R6, R2, R6 ; increment index reg.

Assembler is a program that turns symbols into machine instructions.

- ISA-specific:
  close correspondence between symbols and instruction set
  - mnemonics for opcodes
  - labels for memory locations
- additional operations for allocating storage and initializing data
An Assembly Language Program

; Program to multiply a number by the constant 6

.ORIG x3050
LD R1, SIX
LD R2, NUMBER
AND R3, R3, #0 ; Clear R3. It will contain the product.

; The inner loop

AGAIN ADD R3, R3, R2
ADD R1, R1, #-1 ; R1 keeps track of the iteration.
BRp AGAIN

HALT

NUMBER .BLKW 2
SIX .FILL x0006

.END
Decoding the Label NUMBER for LD instruction

LD R2, NUMBER

Addr = x3050  PC= x3051

Addr for Label “NUMBER” x3059

0010 DR PCOffset

Offset Required = x0008
LC-3 Assembly Language Syntax

Each line of a program is one of the following:

- an instruction
- an assembler directive (or pseudo-op)
- a comment

Whitespace (between symbols) and case are ignored. Comments (beginning with “;”) are also ignored.

Can we write two assembly instructions on the same line?

An instruction has the following format:

```
LABEL   OPCODE   OPERANDS ; COMMENTS
```

- **LABEL**: optional
- **OPCODE**: mandatory
- **OPERANDS**: mandatory
- **; COMMENTS**: mandatory
In Class Exercise

- Do instruction translation by filling in the table below

<table>
<thead>
<tr>
<th>Machine Code</th>
<th>Assembly Code</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001 101 010 1 11111</td>
<td>NOT R5, R2</td>
<td>R5 ← NOT(R2)</td>
</tr>
<tr>
<td>0101 000 011 0 00 001</td>
<td>AND R0, R3, R1</td>
<td>R0 ← R3 AND R1</td>
</tr>
<tr>
<td>0001 110 111 1 00011</td>
<td>ADD R6, R7, #3</td>
<td>R6 ← R7 + SEXT(#3)</td>
</tr>
</tbody>
</table>
Opcodes and Operands

Opcodes
• reserved symbols that correspond to LC-3 instructions
• listed in Appendix A
  ➢ ex: ADD, AND, LD, LDR, …

Operands
• registers -- specified by Rn, where n is the register number
• numbers -- indicated by # (decimal) or x (hex)
• label -- symbolic name of memory location
• separated by comma
• number, order, and type correspond to instruction format
  ➢ ex:

```
ADD  R1,R1,R3  R1 ← R1 + R3  0001 001 001 0 00 011
ADD  R1,R1,#3  R1 ← R1 + #3  0001 001 001 1 00011
LD   R6,NUMBER R6 ← mem[...]  0010 110 xxxxxxxxxxx
BRz  LOOP       If Z, PC ← ...  0000 010 xxxxxxxxxxx
```
Labels and Comments

Label

- placed at the beginning of the line
- assigns a symbolic name to the address corresponding to line

  ```
  LOOP  ADD  R1,R1,#-1
  BRp   LOOP
  ```

Comment

- anything after a semicolon is a comment
- ignored by assembler
- used by humans to document/understand programs

  - tips for useful comments:
    - avoid restating the obvious, as “decrement R1”
    - provide additional insight, as in “accumulate product in R6”
    - use comments to separate pieces of program
Assembler Directives

Pseudo-operations

• do not refer to operations executed by program
• used by assembler
• look like instruction, but “opcode” starts with dot

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ORIG</td>
<td>address</td>
<td>starting address of program</td>
</tr>
<tr>
<td>.END</td>
<td></td>
<td>end of program</td>
</tr>
<tr>
<td>.BLKW</td>
<td>n</td>
<td>allocate n words of storage</td>
</tr>
<tr>
<td>.FILL</td>
<td>n</td>
<td>allocate one word, initialize with value n</td>
</tr>
<tr>
<td>.STRINGZ</td>
<td>n-character string</td>
<td>allocate n+1 locations, initialize w/characters and null terminator</td>
</tr>
</tbody>
</table>
Assembler Directives

.ORIG x3010
HELLO .STRINGZ “Hello, World!”
x3010 : x0048
x3011 : x0065
x3012 : x006C
x3013 : x006C
x3014 : x006f
x3015 : x002C
x3016 : x0020

........
x301C : x0021
x301D : x0000
Trap Codes

LC-3 assembler provides “pseudo-instructions” for each trap code, so you don’t have to remember them.

<table>
<thead>
<tr>
<th>Code</th>
<th>Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT</td>
<td>TRAP x25</td>
<td>Halt execution and print message to console.</td>
</tr>
<tr>
<td>IN</td>
<td>TRAP x23</td>
<td>Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>OUT</td>
<td>TRAP x21</td>
<td>Write one character (in R0[7:0]) to console.</td>
</tr>
<tr>
<td>GETC</td>
<td>TRAP x20</td>
<td>Read one character from keyboard. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>PUTS</td>
<td>TRAP x22</td>
<td>Write null-terminated string to console. Address of string is in R0.</td>
</tr>
</tbody>
</table>
Style Guidelines

Use the following style guidelines to improve the readability and understandability of your programs:

1. Provide a program header, with author’s name, date, etc., and purpose of program.

2. Start labels, opcode, operands, and comments in same column for each line. (Unless entire line is a comment.)

3. Use comments to explain what each register does.

4. Give explanatory comment for most instructions.

5. Use meaningful symbolic names.
   - Mixed upper and lower case for readability.
   - ASCIItoBinary, InputRoutine, SaveR1

6. Provide comments between program sections.

7. Each line must fit on the page -- no wraparound or truncations.
   - Long statements split in aesthetically pleasing manner.
Sample Program

Count the occurrences of a character in a file.
Remember this?

Count = 0  
(R2 = 0)

Ptr = 1st file character  
(R3 = M[x3012])

Input char from keybd  
(TRAP x23)

Load char from file  
(R1 = M[R3])

Done?  
(R1 ?= EOT)

YES

NO

Match?  
(R1 ?= R0)

YES

NO

Incr Count  
(R2 = R2 + 1)

Convert count to ASCII character  
(R0 = x30, R0 = R2 + R0)

Print count  
(TRAP x21)

HALT  
(TRAP x25)

NO

Load next char from file  
(R3 = R3 + 1, R1 = M[R3])
Char Count in Assembly Language (1 of 3)

;;; Program to count occurrences of a character in a file.
;;; Character to be input from the keyboard.
;;; Result to be displayed on the monitor.
;;; Program only works if no more than 9 occurrences are found.

;

; Initialization

.ORIG  x3000
AND   R2, R2, #0   ; R2 is counter, initially 0
LD    R3, PTR      ; R3 is pointer to characters
GETC  ; R0 gets character input

LDR   R1, R3, #0   ; R1 gets first character

; Test character for end of file

TEST  ADD  R4, R1, #-4   ; Test for EOT (ASCII x04)
BRz   OUTPUT        ; If done, prepare the output
Char Count in Assembly Language (2 of 3)

; Test character for match. If a match, increment count.
;
    NOT    R1, R1
    ADD    R1, R1, R0 ; If match, R1 = xFFFF
    NOT    R1, R1    ; If match, R1 = x0000
    BRnp   GETCHAR   ; If no match, do not increment
    ADD    R2, R2, #1

; Get next character from file.
;
GETCHAR ADD    R3, R3, #1 ; Point to next character.
    LDR    R1, R3, #0 ; R1 gets next char to test
    BRnzp  TEST

; Output the count.
;
OUTPUT LD      R0, ASCII ; Load the ASCII template
    ADD    R0, R0, R2 ; Covert binary count to ASCII
    OUT     ; ASCII code in R0 is displayed.
    HALT   ; Halt machine
Char Count in Assembly Language (3 of 3)

; Storage for pointer and ASCII template

ASCII  .FILL  x0030
PTR    .FILL  x4000
.END
Note for HW problems

• Don’t focus on only compiling your program and expect the program to work
  • Think about a good way to test your program
  • Suggestion: write a small script

• To get register values, at the end of the program
  • set breakpoints at HALT instructions
  • use “check r7 x343”

• Read the questions carefully, always look for start of program address

• Address is calculated after .ORIG
  • .ORIG X4545
  • LDI R0,#3
  • Offset is calculated on the PC register value of the current instructions

• LC3 edit does not generate symbol table
• Submit HW in *.bin format
Recap

- **Assembly Language:**
  - Written in the same abstraction as binary instructions
  - More readable
    - `ADD R6, R2, R6 ; increment index reg.`
  - Need an assembler to convert assembly instructions into binary instructions
  - PennSim has in-built assembler
  - Assembly Language can have additional operations for allocating storage and initializing data values
  - In few cases it is possible to write two assembly instructions in the same line
    - LC3 it is possible since LC3 instructions are fixed length
Recap

Each line of a program is one of the following:

- an instruction
- an assembler directive (or pseudo-op)
- a comment

Whitespace (between symbols) and case are ignored. Comments (beginning with “;”) are also ignored.

An instruction has the following format:

```
LABEL OPCODE OPERANDS ; COMMENTS
```

- `LABEL` is optional.
- `OPCODE`, `OPERANDS`, and `COMMENTS` are mandatory.
Recap: Trap Codes

LC-3 assembler provides “pseudo-instructions” for each trap code, so you don’t have to remember them.

<table>
<thead>
<tr>
<th>Code</th>
<th>Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT</td>
<td>TRAP x25</td>
<td>Halt execution and print message to console.</td>
</tr>
<tr>
<td>IN</td>
<td>TRAP x23</td>
<td>Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>OUT</td>
<td>TRAP x21</td>
<td>Write one character (in R0[7:0]) to console.</td>
</tr>
<tr>
<td>GETC</td>
<td>TRAP x20</td>
<td>Read one character from keyboard. Character stored in R0[7:0].</td>
</tr>
<tr>
<td>PUTS</td>
<td>TRAP x22</td>
<td>Write null-terminated string to console. Address of string is in R0.</td>
</tr>
</tbody>
</table>
Assembly Process

Convert assembly language file (.asm) into an executable file (.obj) for the LC-3 simulator.

First Pass:
- scan program file
- find all labels and calculate the corresponding addresses; this is called the *symbol table*

Second Pass:
- convert instructions to machine language, using information from symbol table
First Pass: Constructing the Symbol Table

1. **Find the `.ORIG` statement,**
   which tells us the address of the first instruction.
   - Initialize location counter (LC), which keeps track of the current instruction.

2. **For each non-empty line in the program:**
   a) If line contains a label, add label and LC to symbol table.
   b) Increment LC.
      - **NOTE:** If statement is `.BLKW` or `.STRINGZ`, increment LC by the number of words allocated.

3. **Stop when `.END` statement is reached.**

**NOTE:** A line that contains only a comment is considered an empty line.
Practice

Construct the symbol table for the program in Figure 7.1 (Slides 7-15 through 7-17).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>x3004</td>
</tr>
<tr>
<td>getchar</td>
<td>x300B</td>
</tr>
<tr>
<td>output</td>
<td>x300E</td>
</tr>
<tr>
<td>ASCII</td>
<td>x3012</td>
</tr>
<tr>
<td>PTR</td>
<td>x3013</td>
</tr>
</tbody>
</table>
In-Class Exercise (2014 Exam4 Question2b)

Construct the symbol table for the program.

```
.ORIG x3000
LEA R2, STRING
LD R3, NUMBER
HERE ADD R1, R2, R3
ADD R2, R1, #0
LDR R0, R1, #0
BRz DONE
OUT
BR HERE
THIS .BLKW 6
STRING .STRINGZ "2down_3to_go"
NUMBER .FILL x4
DONE HALT
.END
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>HERE</td>
<td>x3002</td>
</tr>
<tr>
<td>THIS</td>
<td>x3008</td>
</tr>
<tr>
<td>STRING</td>
<td>x300E</td>
</tr>
<tr>
<td>NUMBER</td>
<td>x301B</td>
</tr>
<tr>
<td>DONE</td>
<td>x301C</td>
</tr>
</tbody>
</table>
Second Pass: Generating Machine Language

For each executable assembly language statement, generate the corresponding machine language instruction.

- If operand is a label,
  look up the address from the symbol table.

Potential problems:

- Improper number or type of arguments
  
  - ex: NOT R1, #7
  - ADD R1, R2
  - ADD R3, R3, NUMBER

- Immediate argument too large
  
  - ex: ADD R1, R2, #1023

- Address (associated with label) more than 256 from instruction
  
  - can’t use PC-relative addressing mode
Practice

Identify and correct 5 assembly errors in the following LC3 program:

```
.ORIG x3001
AND R4, R4, #0
BRz NEXT
LD R5, STRING
STOP
ADD R5, R5, *1
BRp STOP
LDR R6, R5, #4
OR R6, R6, #3
ST R4, STRING
STOP
HALT
ZERO
.VAL
.STRING
```

.END

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
Practice

Identify and correct 5 assembly errors in the following LC3 program:

```
.ORIG x3001
AND R4, R4, #0
BRz NEXT
LD R5, STRING
STOP
ADD R5, R5, *1
BRp STOP
LDR R6, R5, #4
OR R6, R6, #3
ST R4, STRING
STOP
HALT
ZERO
.VAL .FILL #0
STRING .STRINGS "Hello World !"
.END
```

- OR Instruction not part of LC3
Practice

Identify and correct 5 assembly errors in the following LC3 program:

```
.ORIG x3001
AND R4, R4, #0
BRz NEXT
LD R5, STRING
STOP
ADD R5, R5, *1
BRp STOP
LDR R6, R5, #4
OR R6, R6, #3
ST R4, STRING
STOP
HALT
ZERO
.FILL #0
VAL
.FILL VALUE
STRING
.STRINGS "Hello World !"
.END
```
Practice

Identify and correct 5 assembly errors in the following LC3 program:

```
.ORIG x3001
AND R4, R4, #0
BRz NEXT
LD R5, STRING
STOP
ADD R5, R5, *1
BRp STOP
LDR R6, R5, #4
OR R6, R6, #3
ST R4, STRING
STOP
HALT
ZERO
.FILL #0
VAL
.FILL VALUE
STRING
.STRINGS "Hello World !"
.END
```

Immediate Value using *
Practice

Identify and correct 5 assembly errors in the following LC3 program:

```assembly
.ORIG x3001
AND R4, R4, #0
BRz NEXT
LD R5, STRING
STOP
ADD R5, R5, *1
BRp STOP
LDR R6, R5, #4
OR R6, R6, #3
ST R4, STRING
STOP
HALT
ZERO
.FILL #0
VAL
.FILL VALUE
STRING
.STRINGS "Hello World !"
.END
```

- NEXT label undeclared
- undeclared
Practice

Identify and correct 5 assembly errors in the following LC3 program:

```
.ORIG x3001
AND R4, R4, #0
BRz NEXT
LD R5, STRING
ADD R5, R5, *1
BRp STOP
LDR R6, R5, #4
OR R6, R6, #3
ST R4, STRING
HALT
ZERO
VAL
STRING
```
Practice

Identify and correct 5 assembly errors in the following LC3 program:

```
.ORIG x3001
AND R4, R4, #0
BRz NEXT
LD R5, STRING
STOP
ADD R5, R5, *1
BRp STOP
LDR R6, R5, #4
OR R6, R6, #3
ST R4, STRING
STOP
HALT
ZERO .FILL #0
VAL .FILL VALUE
STRING .STRINGS "Hello World !"
.END
```

VALUE label undeclared
Using the symbol table constructed earlier, translate these statements into LC-3 machine language.

<table>
<thead>
<tr>
<th>Address</th>
<th>Statement</th>
<th>Machine Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3001</td>
<td>LD R3, PTR</td>
<td>0010 011 000010001 = x2611</td>
</tr>
<tr>
<td>x3004</td>
<td>ADD R4, R1, #−4</td>
<td>0001 100 0011 11100 = x187C</td>
</tr>
<tr>
<td>x300C</td>
<td>LDR R1, R3, #0</td>
<td>0110 001 011 000000 = x62C0</td>
</tr>
<tr>
<td>x3009</td>
<td>BRnp GETCHAR</td>
<td>0000 101 000000001 = x0A01</td>
</tr>
</tbody>
</table>

Symbol Table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>x3004</td>
</tr>
<tr>
<td>GETCHAR</td>
<td>x300B</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>x300E</td>
</tr>
<tr>
<td>ASCII</td>
<td>x3012</td>
</tr>
<tr>
<td>PTR</td>
<td>x3013</td>
</tr>
</tbody>
</table>
In-Class Exercise

Construct the symbol table for the program.

```
.ORIG x3000
LD R2, NUMBER
LD R1, MASK
LD R3, PTR2
LOOP LDR R4, R3, #0
     AND R4, R4, R1
     BRz NEXT
     ADD R0, R0, #1
NEXT ADD R3, R3, #1
     ADD R2, R2, #-1
     BRp LOOP
     STI R0, PTR1
     HALT
NUMBER .BLKW 3
MASK .FILL x8000
PTR1 .FILL x4000
PTR2 .FILL x5000
.END
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>x3003</td>
</tr>
<tr>
<td>NEXT</td>
<td>x3007</td>
</tr>
<tr>
<td>NUMBER</td>
<td>x300B</td>
</tr>
<tr>
<td>MASK</td>
<td>x300E</td>
</tr>
<tr>
<td>PTR1</td>
<td>x300F</td>
</tr>
<tr>
<td>PTR2</td>
<td>x3010</td>
</tr>
</tbody>
</table>
Using the symbol table constructed earlier, translate these statements into LC-3 machine language.

<table>
<thead>
<tr>
<th>Address</th>
<th>Statement</th>
<th>Machine Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>x3000</td>
<td>ADD R1,R2,#4</td>
<td>0001 001 010 1 00100 = x12A4</td>
</tr>
<tr>
<td>x300A</td>
<td>STI R0,PTR3</td>
<td>1011 000 000 0 01010 = xB00A</td>
</tr>
</tbody>
</table>

**Symbol Table**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS</td>
<td>x3012</td>
</tr>
<tr>
<td>AGAIN</td>
<td>x3014</td>
</tr>
<tr>
<td>PTR3</td>
<td>x3015</td>
</tr>
<tr>
<td>DESTINATION</td>
<td>x301A</td>
</tr>
</tbody>
</table>
Note for All

• HW6 due today
  • Need to submit your **binary code** at Learn@uw, by 9.55am
  • Also need to hand in physical copy
  • Remember to staple your HW

• HW7 will be released today
  • Due on April 14th
Points covered so far

• Assembly Language
  • Structure
  • Labels
  • Assembler directives

• Two step assembly process
  • generating the symbol table
  • converting assembly to machine instructions
LC-3 Assembler

Using “lc3as” (Unix) or LC3Edit (Windows), generates several different output files.

Assembly Language Program (.asm) → Assembler → Binary Listing (.bin) → Object File (.obj)
Hex Listing (.hex)
Symbol Table (.sym)
Listing File (.lst)

This one gets loaded into the simulator.
Object File Format

LC-3 object file contains

• Starting address (location where program must be loaded), followed by…
• Machine instructions

Example

• Beginning of “count character” object file looks like this:

```
0011000000000000
0101010010100000
0010011000010001
1111000000100011
```

```
.ORIG x3000
AND R2, R2, #0
LD R3, PTR
TRAP x23
```
Multiple Object Files

An object file is not necessarily a complete program.

- system-provided library routines
- code blocks written by multiple developers

For LC-3 simulator, can load multiple object files into memory, then start executing at a desired address.

- system routines, such as keyboard input, are loaded automatically
  - loaded into “system memory,” below x3000
  - user code should be loaded between x3000 and xFDFF
- each object file includes a starting address
- be careful not to load overlapping object files
Linking and Loading

*Loading* is the process of copying an executable image into memory.

- more sophisticated loaders are able to *relocate* images to fit into available memory
- must readjust branch targets, load/store addresses

*Linking* is the process of resolving symbols between independent object files.

- suppose we define a symbol in one module, and want to use it in another
- some notation, such as `.EXTERNAL`, is used to tell assembler that a symbol is defined in another module
- linker will search symbol tables of other modules to resolve symbols and complete code generation before loading
Linking

P1:
.ORIG x3000
ADD R0, R1, #0
ADD R0, R0, #-15
ADD R0, R0, #-10
STI R0, PTR
HALT
.END

P2:
PTR .FILL x5000
.END
Linking

P1:
.ORIG x3000
ADD R0, R1, #0
ADD R0, R0, #-15
ADD R0, R0, #-10
STI R0, PTR
HALT
.EXTERNAL PTR
.END

P2:
PTR .FILL x5000
.END
Skipping Ahead to Chapter 9

You will need to use subroutines for programming assignments

• Read Section 9.2

A subroutine is a program fragment that:

• performs a well-defined task
• is invoked (called) by another user program
• returns control to the calling program when finished

Reasons for subroutines:

• reuse useful (and debugged!) code without having to keep typing it in
• divide task among multiple programmers
• use vendor-supplied library of useful routines
Subroutine Motivation

Reasons for subroutines:
- reuse useful code without having to keep typing it in
- divide task among multiple programmers

Figure 9.7  Instruction execution flow with/without subroutines
JSR Instruction

Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.

- saving the return address is called “linking”
- target address is PC-relative (PC + Sext(IR[10:0]))
- bit 11 specifies addressing mode
  - if =1, PC-relative: target address = PC + Sext(IR[10:0])
  - if =0, register: target address = contents of register IR[8:6]
NOTE: PC has already been incremented during instruction fetch stage.
JSR Example
PC is currently x4200

What is the contents of R7 and PC after the following instruction is executed?

```
<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

NOTE: PC has already been incremented during instruction fetch stage.

1. 0x4201
2. 0x4201
3. 0x3E05

0000 0100 = 0x404
1111 1100 0000 0100 = 0xFC04

JSR label; Jump to Subroutine

R7 ← PC', PC ← PC' + Sext(PCoffset11)

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.
JSRR Instruction

Just like JSR, except Register addressing mode.
  • target address is Base Register
  • bit 11 specifies addressing mode

What important feature does JSRR provide that JSR does not?
Subroutine (target) address can be anywhere in memory
NOTE: PC has already been incremented during instruction fetch stage.
JSRR Example

PC is currently x420A

R5 is currently x3002

What is the contents of R7 and PC after the following instruction is executed?

```
JSRR 01000000
      01000000 10100000
```

NOTE: PC has already been incremented during instruction fetch stage.
Returning from a Subroutine

RET (JMP R7) gets us back to the calling routine.
• just like TRAP
Example: Negate the value in R0

2sComp  NOT  R0, R0  ; flip bits
        ADD  R0, R0, #1  ; add one
        RET  ; return to caller

To call from a program (within 1024 instructions):

; need to compute R4 = R1 - R3
        ADD  R0, R3, #0  ; copy R3 to R0
        JSR  2sComp  ; negate
        ADD  R4, R1, R0  ; add to R1
        ... 

Note: Caller should save R0 if we’ll need it later!
Passing Information to/from Subroutines

Arguments

• A value passed in to a subroutine is called an argument.
• This is a value needed by the subroutine to do its job.
• Examples:
  ➢ In 2sComp routine, R0 is the number to be negated
  ➢ In OUT service routine, R0 is the character to be printed.
  ➢ In PUTS routine, R0 is address of string to be printed.

Return Values

• A value passed out of a subroutine is called a return value.
• This is the value that you called the subroutine to compute.
• Examples:
  ➢ In 2sComp routine, negated value is returned in R0.
  ➢ In GETC service routine, character read from the keyboard is returned in R0.
Using Subroutines

In order to use a subroutine, a programmer must know:

• its address (or at least a label that will be bound to its address)
• its function (what does it do?)
  ➢ NOTE: The programmer does not need to know how the subroutine works, but what changes are visible in the machine’s state after the routine has run.
• its arguments (where to pass data in, if any)
• its return values (where to get computed data, if any)
Saving and Restore Registers

Since subroutines are just like service routines, we also need to save and restore registers, if needed.

Generally use “callee-save” strategy, except for return values.

• Save anything that the subroutine will alter internally that shouldn’t be visible when the subroutine returns.
• It’s good practice to restore incoming arguments to their original values (unless overwritten by return value).

**Remember:** You MUST save R7 if you call any other subroutine or service routine (TRAP).
• Otherwise, you won’t be able to return to caller.
Example

(1) Write a subroutine **FirstChar** to:

find the first occurrence
of a particular character (in R0)
in a string (pointed to by R1);
return pointer to character or to end of string (NULL) in R2.

(2) Use FirstChar to write **CountChar**, which:

counts the number of occurrences
of a particular character (in R0)
in a string (pointed to by R1);
return count in R2.

Can write CountChar subroutine using FirstChar
Alternatively, can write the second subroutine first,
without knowing the implementation of FirstChar!
**FirstChar Algorithm**

1. **save regs**
2. **R2 <- R1**
3. **R3 <- M(R2)**
4. **R3=0**
5. **R3=R0**
6. **R2 <- R2 + 1**
7. **restore regs**
8. **return**

- **R0** holds character to search
- **R1** is pointer to string
- **R2** is return value
- Pointer to character or end of string
FirstChar Implementation

; FirstChar: subroutine to find first occurrence of a char

FirstChar

ST    R3, FCR3 ; save registers
ST    R4, FCR4
NOT   R4, R0 ; use R4 for comparisons
ADD   R4, R4, #1
ADD   R2, R1, #0 ; initialize ptr to beginning of string

FC1  LDR    R3, R2, #0 ; read character
BRz   FC2 ; if null, we’re done
ADD   R3, R3, R4 ; see if matches input char
BRz   FC2 ; if yes, we’re done
ADD   R2, R2, #1 ; increment pointer
BRnzp FC1

FC2  LD     R3, FCR3 ; restore registers
LD     R4, FCR4
RET    ; and return
CountChar Algorithm (using FirstChar)

1. save regs
2. call FirstChar
3. R3 <- M(R2)
4. R1 <- R2 + 1
5. restore regs
6. return

- R0 holds character to search
- R1 is pointer to string
- R2 is return value
- count of occurrences

R7 needs to be saved since we’re using JSR
CountChar Implementation

;  CountChar: subroutine to count occurrences of a char

CountChar

ST  R3, CCR3  ;  save registers
ST  R4, CCR4
ST  R7, CCR7  ;  JSR alters R7
ST  R1, CCR1  ;  save original string ptr
AND R4, R4, #0  ;  initialize count to zero

CC1  JSR  FirstChar  ;  find next occurrence (ptr in R2)
LDR R3, R2, #0  ;  see if char or null
BRz  CC2  ;  if null, no more chars
ADD R4, R4, #1  ;  increment count
ADD R1, R2, #1  ;  point to next char in string
BRnzp CC1

CC2  ADD R2, R4, #0  ;  move return val (count) to R2
LD  R3, CCR3  ;  restore regs
LD  R4, CCR4
LD  R1, CCR1
LD  R7, CCR7
RET  ;  and return
Subroutine Summary

- Subroutines are useful reuse of code
- Need address, function, arguments, and return value
- Use JSR, JSRR to call subroutine
- Use RET to return from subroutine

- PennSim Demo

- If the return address is written to R7, then can we call a subroutine within a subroutine?
  
  Save R7

- Can we make recursive calls?
  
  Stack, discussed in Chapter 10, but not this course.