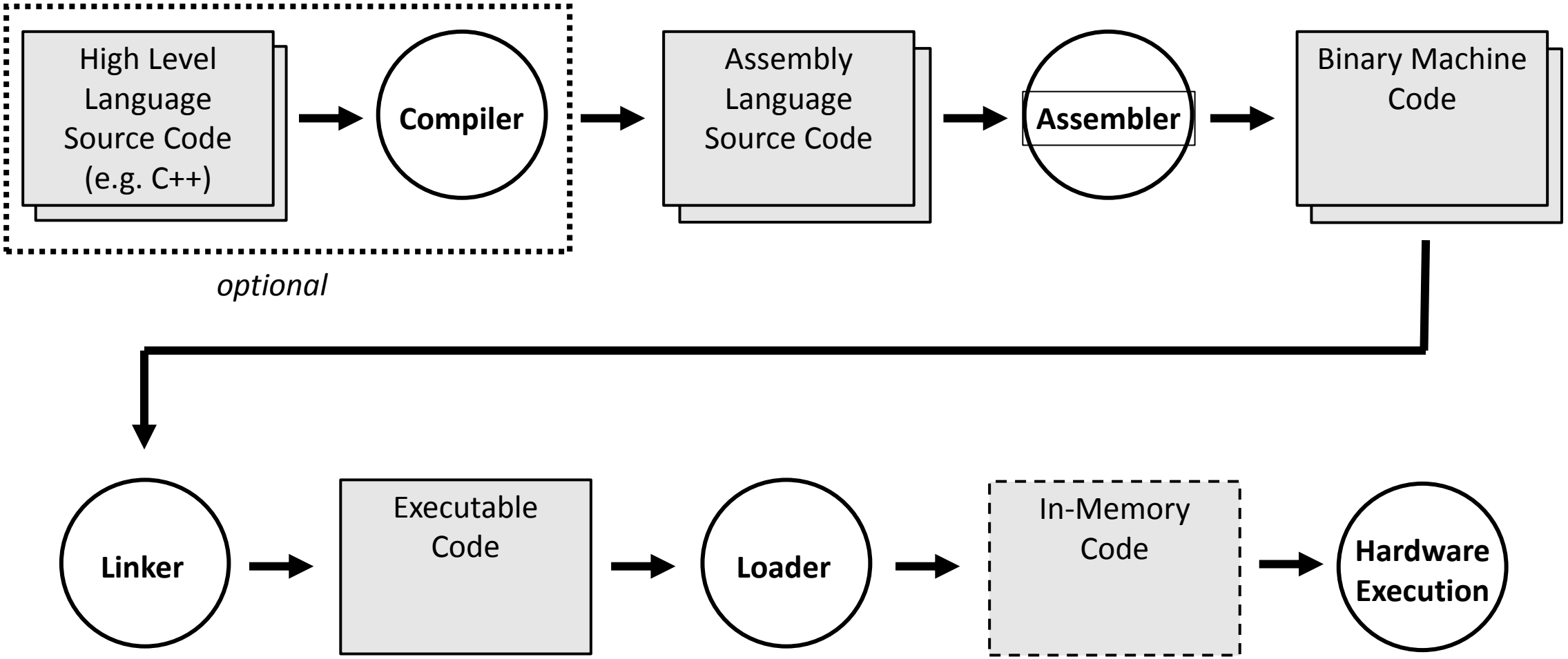
CMPE 220

Class 7– Macro Processing

Building Software



What is a “Macro?”

- A *macro* (which stands for "macroinstruction") is a programmable pattern which translates a certain sequence of input into a preset sequence of output.
  - A *macro definition* defines an input sequence, and the corresponding output sequence (*expansion*)
  - Using a macro within an input stream is called an *invocation*
- Macros can be used to make programming (or other tasks) less repetitive.
- Macros are another step in the direction of *convenience*

Example

- Macro Definition (syntax will vary depending on macro language):
  aliqua :: bananas are rich in potassium ;;

Input with *Macro Invocations*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna %aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse %aliqua dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Output with *Macro Expansions*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna bananas are rich in potassium.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse bananas are rich in potassium dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Example: Positional Arguments

- Macro Definition (syntax will vary depending on macro language):
  fruit() :: %1 are rich in %2 and %3 ;;

Input with *Macro Invocations*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna %fruit(bananas, potassium, Vitamin B6).

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse %fruit(oranges, Vitamin C, Thiamin) dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Output with *Macro Expansions*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna bananas are rich in potassium and Vitamin B6.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse oranges are rich in Vitamin C and Thiamin dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Example: *Named* Arguments with Defaults

- Macro Definition (syntax will vary depending on macro language):
  fruit(%name, %nutrient1, %nutrient2=sugar) ::
  %name are rich in %nutrient1 and %nutrient2 ;;

Input with *Macro Invocations*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna %fruit(name=bananas, nutrient1=potassium, nutrient2=Vitamin B6).

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse %fruit(name= oranges, nutrient1= Vitamin C) dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Output with *Macro Expansions*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna bananas are rich in potassium and Vitamin B6.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse oranges are rich in Vitamin C and sugar dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Example: *Named* Arguments with Defaults

- Macro Definition (syntax will vary depending on macro language):
  fruit(%name, %nutrient1, %nutrient2=sugar) ::
  %name are rich in %nutrient1 and %nutrient2 ;;

- Omitting an argument that has a default is legal; the default value is used.
  I've heard that %fruit(%name=bananas, %nutrient1=potassium).

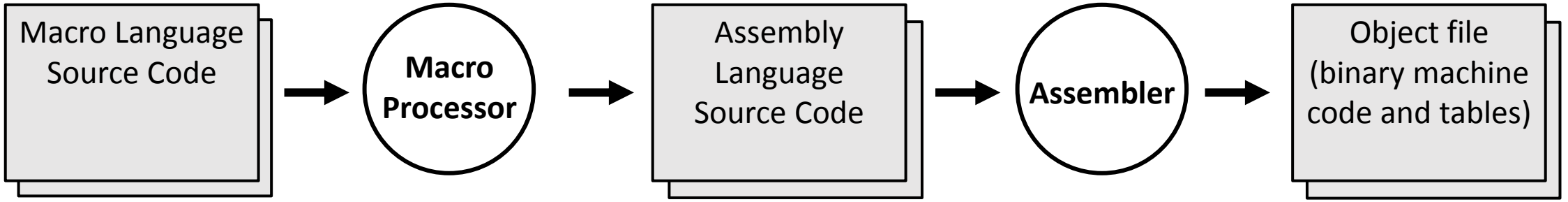
I've heard that bananas are rich in potassium and sugar.

- Omitting an argument that does *not* have a default is an error.
  I've heard that %fruit(%name=bananas, %nutrient2=Vitamin C)

ERROR – missing %nutrient1

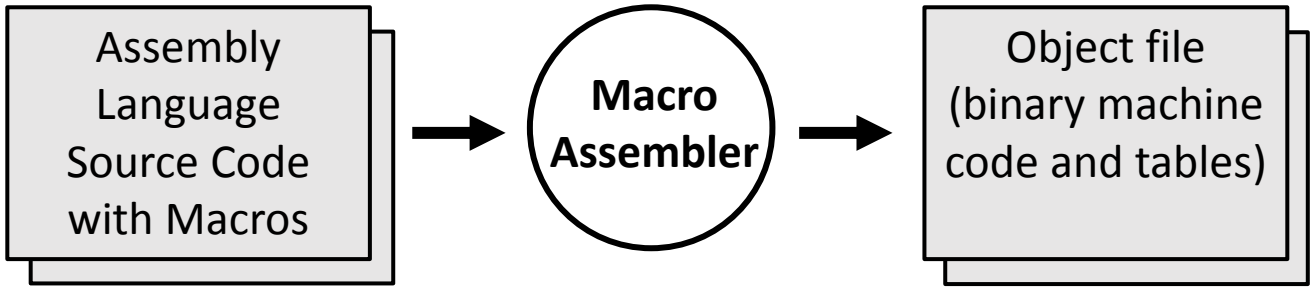
Incorporating Macros

- A macro-processor may be an standalone program that processes text before it is submitted to a compiler or assembler:



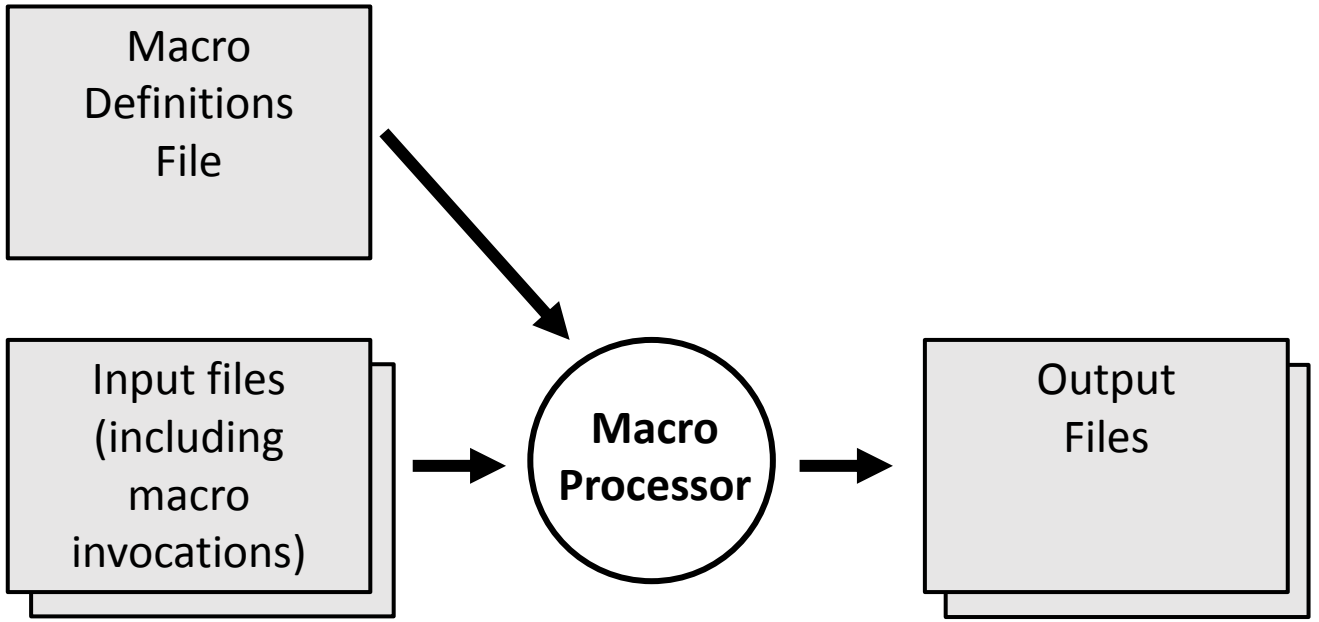
Incorporating Macros

- Macro-processing can be built into a compiler or an assembler (i.e. a *macro assembler*)



Separating Definitions from Input Files

- Macro definitions can reside in separate files, allowing shared "libraries" of macro definitions



Macros Are Not Limited to Programming

- Suppose our company produces documents for using medical equipment. Out attorneys want us to include some legal *disclaimers*, but the disclaimer language sometimes changes.

Input with *Macro Invocations*

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse aliento dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

%disclaimer

Macros Do Not Extend Instruction Sets!

- Macros simply translate one text pattern into another.
- When used with assembly language – whether as a pre-processor, or a macro assembler – they do not add new instructions to the machine.
- **Caution:** A single macro may expand to many instructions. If the macro is used frequently, it can cause code size to balloon.
- **Caution:** Macros may also have hidden side effects, such as changing register values or the Condition Code (CC)
  - Programmers who use macros need to keep these side effects in mind

Macro Versus Subroutines

Macro	Subroutine
Code is duplicated each time the macro is invoked	Only one copy of the subroutine code
Labels can be a problem; if a label is used within a macro, it will be duplicated each time the macro is invoked, generating a duplicate label error	Labels can be freely used within the subroutine
Macros may allow sophisticated argument handling, e.g. changing order, default values, etc.	Argument handling is limited by the language. Most assemblers have no provisions for arguments

Real-World Use Case

- I was once assigned to test a math library on a new computer
  - SIN, COS, TAN, ASIN, ACOS, ATAN, LOG, etc

For each function, I needed to:

- Test if the function returned known correct values for a range of arguments
- Test the “sensitivity” to see if return values changed when input arguments were varied by the smallest machine precision
- Test error handling to ensure an error status was return for invalid arguments

Writing the test cases was horribly repetitive and tedious!

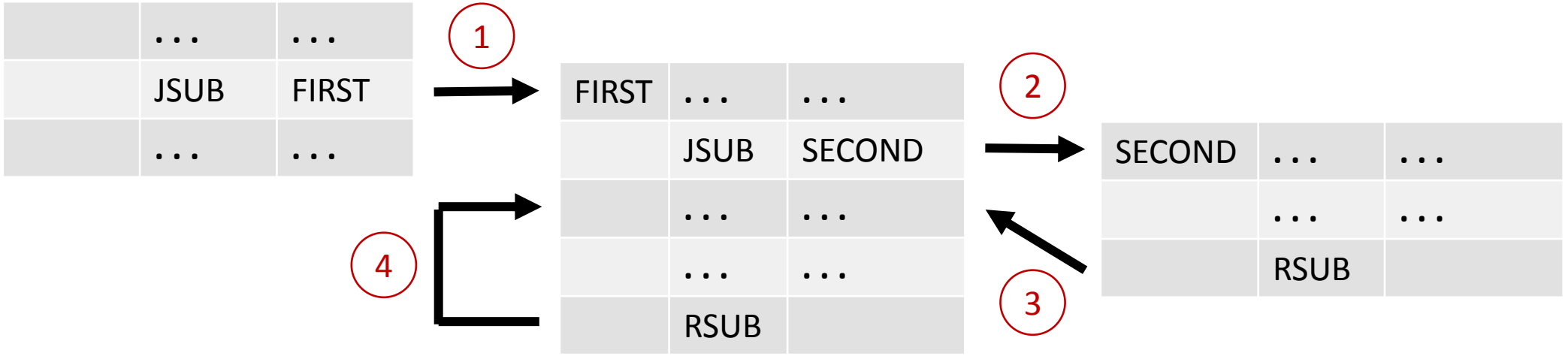
Macros Reduced the Coding to 1 Line Per Test

- Sample Macro Definition (using pseudocode)

```
1  ERRORTEST (%function, %value) ::
2  IF (%function (%value) NE NaN) THEN
3      print ”Function %function failed to return an NaN for argument %value)
4  ENDIF
5  ;;
```

Example: a Better JSUB

- The SIC instruction set has a subroutine call – JSUB
- Because there is only one return address register (L), subroutines cannot be nested:



Macro Subroutine Call

- Our Macro Subroutine Call will use a *stack* to store the return address, allowing subroutines to be nested
- We will need three macros:
- **SINIT:** Set up and initialize the subroutine return address stack
- **SJMP:** Subroutine jump
- **SRET:** Subroutine return

SINIT: Initialize Subroutine Stack

1	SINIT ::			
2	SSTACK	RESW	10	Allow 10 nested subroutine calls
3	SPTR	WORD	#SSTACK	Stack pointer – contains the address of the stack
4	SAVEA	RESW	1	Space to save A register (we'll see why)
5	;;			

- SINIT needs to be invoked in the data area of our program file
- **Hidden Risk:** If we nest subroutines more than 10 deep, our stack will "overflow" and SJMP and SRET will behave unpredictably.

SJMP: Subroutine Jump

1	SJMP(%ADDR) ::			
2		JSUB	*+3	Get address into L register
3		STA	SAVEA	Save the A register
4		RMO	L, A	Move address into A register
5		ADD	24	Increment address to point *past* our macro
6		STA	@SPTR	Use indirect addressing to store return address on stack
7		LDA	SPTR	Increment stack pointer
8		ADD	3	
9		STA	SPTR	
10		LDA	SAVEA	Restore the A register
11		J	%ADDR	Jump to the subroutine
12	;;			

SRET: Subroutine Return

1	SRET ::			
2		STA	SAVEA	Save the A register
3		LDA	SPTR	Decrement stack pointer
4		SUB	3	
5		STA	SPTR	
6		LDA	SAVEA	Restore the A register
7		LDL	@SPTR	Indirect load to get the return address into L register
8		RSUB		Return
9	;;			

Downsides of the Macro Subroutine Call

- Normally a SIC subroutine call costs two instructions
  - JSUB: 1 instruction for each call
  - RSUB: 1 instruction for the return from the subroutine
- Our Macro Subroutine call costs 17 instructions
  - SJMP: 10 instructions for each call
  - SRET: 7 instructions for the return from the subroutine
- There is a hidden risk of a stack overflow
  - We could eliminate the hidden risk of a stack overflow by adding code in the JSUB macro to check for more than 10 saved addresses, and jump to an error routine – but this would add more code

Additional Features

Because – as we've seen – nothing stays simple!

Escape Characters

- Macro languages usually make use of special characters
  - Our sample language uses the ‘%’ character
- If the target language we are generating uses that special character, we have a problem, because the macro processor will be confused when it sees the character in the macro definition
- We use an escape character – usually ‘\’ – to tell the macroprocessor to simply pass through the next character and not treat it as part of the macro language
  - \%

Automatic Label Generation

- Labels can be included within macros without duplication
- The macro processor assigns a unique label each to the macro is invoked – usually based on a simple counter

Defining Automatic Labels

Original SJMP

1	SJMP(%ADDR) ::		
2		JSUB	*+3
3		STA	SAVEA
4		RMO	L, A
5		ADD	24
6		STA	@SPTR
7		LDA	SPTR
8		ADD	3
9		STA	SPTR
10		LDA	SAVEA
11		J	%ADDR
12	;;		

SJMP with Automatic Labels

1	SJMP(%ADDR) ::		
2		STA	SAVEA
3		LDA	#END%%
4		ADD	3
5		STA	@SPTR
6		LDA	SPTR
7		ADD	3
8		STA	SPTR
9		LDA	SAVEA
10	END%%	J	%ADDR
11	;;		

Using Automatic Labels

First Time Macro is Invoked

1		STA	SAVEA
2		LDA	#END01
3		ADD	3
4		STA	@SPTR
5		LDA	SPTR
6		ADD	3
7		STA	SPTR
8		LDA	SAVEA
9	END01	J	%ADDR
10	;;		

Second Time Macro is Invoked

1		STA	SAVEA
2		LDA	#END02
3		ADD	3
4		STA	@SPTR
5		LDA	SPTR
6		ADD	3
7		STA	SPTR
8		LDA	SAVEA
9	END02	J	%ADDR
10	;;		

Conditional Macros

- Macro Processors may add “conditional” statements to generate different code when the macro is invoked.
- The conditional statements are *processed* when the macro is invoked… they are not part of the generated code

Conditional Macro - Example

1	SJMP(%ADDR, %SAVE=YES) ::		
2	%IF (%SAVE EQ YES)		
3		STA	SAVEA
4	%ENDIF		
5		LDA	#END%%
6		ADD	3
7		STA	@SPTR
8		LDA	SPTR
9		ADD	3
10		STA	SPTR
11	%IF (%SAVE EQ YES)		
12		LDA	SAVEA
13	%ENDIF		
14	END%%	J	%ADDR
15	;;		

Conditionally Generated Code

SJMP(%ADDR=MYLABEL, %SAVE=YES)

1		STA	SAVEA
2		LDA	#END01
3		ADD	3
4		STA	@SPTR
5		LDA	SPTR
6		ADD	3
7		STA	SPTR
8		LDA	SAVEA
9	END01	J	MYLABEL
10	;;		

SJMP(%ADDR=MYLABEL, %SAVE=NO)

1		LDA	#END02
2		ADD	3
3		STA	@SPTR
4		LDA	SPTR
5		ADD	3
6		STA	SPTR
7	END02	J	MYLABEL
8	;;		

Macro Variables

- Allows us to define and use variables within the macro processor
- These variables have nothing to do with the target language. They are visible only to the macro processor

Macro Variables - Example

- We can define a variable and set its value

```
1   DEFINE %DEBUG = YES ;;
```

- We can then use that variable within macro definitions

```
1   MYMACRO ::
2
3
4   %IF (%DEBUG EQ YES)
5                          (some special debugging code for our final program)
6   %ENDIF
7
8
9   ;;
```

Nested Macros

- With a simple, single-pass macroprocessor, a macro definition cannot contain a macro invocation
- There are two ways to allow a macro definition to contain a macro invocation:
  - iteration
  - recursion

Single Pass Macro Processing

Pseudocode

1	read source file into input	
2	status = macroprocessor(input, output)	Process the input, searching for macro invocations, and return the output with expansion
3	Write output to output file	

- Does not support nested macros

Macro Processing With Iteration

Pseudocode

1	read source file into input	
2	WHILE (macroprocessor(input, output) EQ invocations_found)	
3	input = output	
4	ENDWHILE	
5	Write output to output file	

- Additional passes, as long as there are nested macros
- May require many passes over the code
- Requires input and output to be held in memory

Macro Processing with Recursion

What is Recursion?

- The process in which a function calls itself directly or indirectly is called *recursion* and the corresponding function is called as *recursive function*.
- Recursion depends on a *stack*, which means that each time a function is called, it has its own copy of local variables, and its own return address

Macro Processing With Recursion

Pseudocode

```
1   read source file into input
2   macroprocessor(input)
3   Write output to output file
```

- Single pass, but depends on the ability of macroprocessor() to call itself recursively
- Minimizes the amount of data held in memory

```
1   FUNCTION macroprocessor(input)
2      output = “”
3      WHILE ( token = getnexttoken(input) )
4         IF (token is a macro invocation)
5            expansion = expandmacro(token)
6            macroprocessor(expansion)
7         ELSE
8            write token to output file
9         ENDIF
10     ENDWHILE
11  RETURN
```

Built-In Macro Processing

- Many assemblers and high-level language compilers feature built-in macro processing features
- Macro Pre-Processor
  - Expands macros into a separate file which is then passed to the compiler or assembler
- Built-In Macro Processor
  - Typically processes each line of the input, and then passes the expanded lines to the scanner
  - Best know example is and ANSI C compiler, which includes the ability to define and expand macros

Steps in Translating a Language

- Early assemblers used “brute force” techniques for scanning the input stream.
  - That was workable, because the structure of assembly language is very simple:
    label opcode operands comment
- Brute force techniques don’t work for high-level languages
- Macro Processors fall somewhere in between
- When we talk about **compilers**, we’ll break out the software components that are used to break down complex languages

Writing a Macro Processor

Software Components

- A *scanner* to search the input for macro definitions and invocations
- A macro and variable *lookup* routine
- A Variable table & lookup routine
- A routine to *emit* the macro expansion

For Next Week

- Log in to Canvas and complete Assignment 4 – Macros