
Version 2.14
UCSD CSE 30 Section B
Computer Organization and Systems Programming
Lecture 2
Keith Muller
DEC PDP 11/45 - 1973

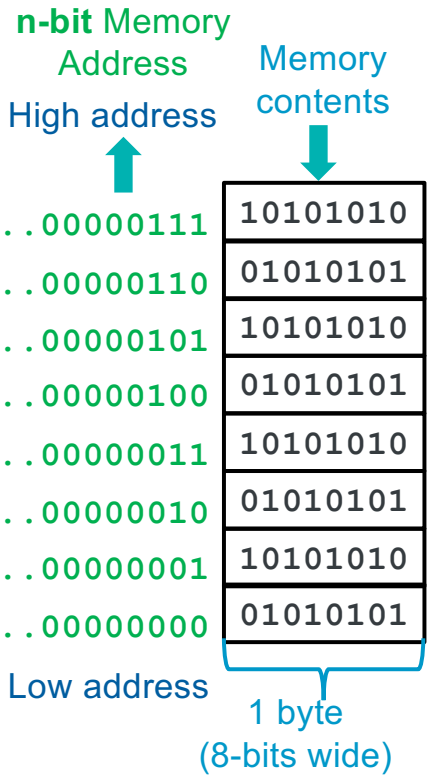
Attendance Code

Memory is Organized in Units of Bytes

- One bit (digit) of storage (in memory) has two possible **states**: 0 or 1
- Memory is organized into a **fixed unit** of 8 bits, called a **byte**

MSB Most Significant Bit →	Bit position in Byte								← LSB (Least Significant Bit)
	7	6	5	4	3	2	1	0	
	0	0	1	0	0	0	0	0	
	Content of each bit								

- Conceptually, memory is a single, **large array** of **bytes**, **where each byte** has a unique *address (byte addressable memory)*
- An address is an **unsigned** (positive #) *fixed-length* n-bit binary value
  - Range (domain) of possible addresses = *address space*
- Each byte in memory can be **individually accessed** and operated on given its **unique address**
- **Word size: is the number of bits in an address**

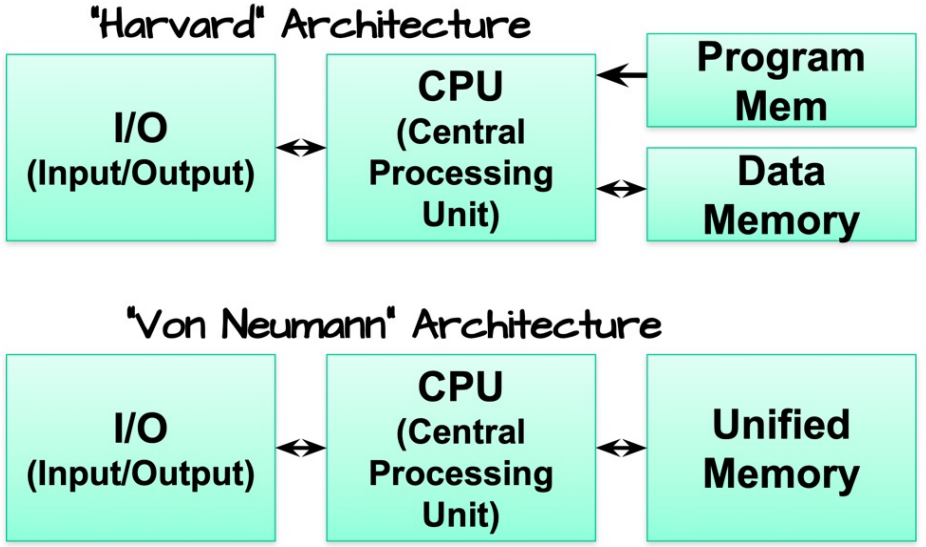


2^N *Bytes of memory*

x

An Alternative that was not successful: Harvard Architecture

- **Harvard architecture premise:** Instructions and data should not interact (claim is higher higher performance), and they can have different word sizes
  - **Observation:** Two memory subsystems (using similar state of the art technologies) can be accessed concurrently for higher throughput
- **Distinguishing feature**: **Independent** instruction and data memories
- Do you agree and why?

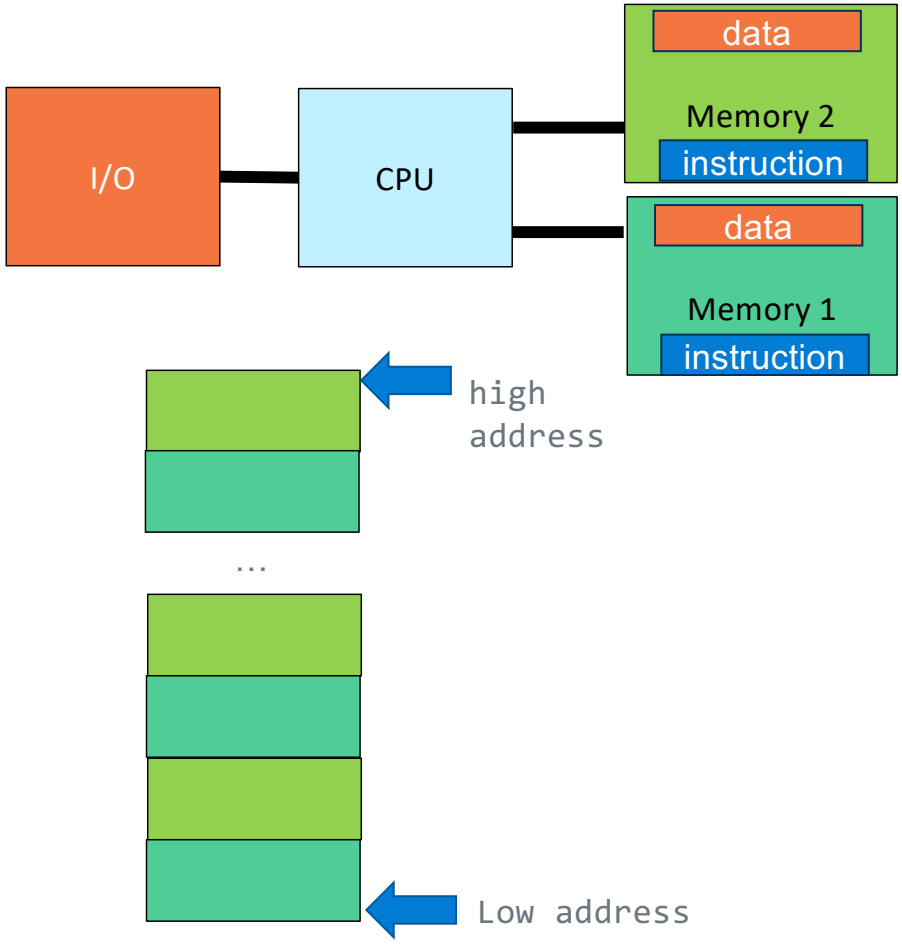


x

Machine Organization Example – Which Architecture is it?

- A good exam question
- Answer: Either you must be told where the Instructions and data are placed
- How can this be a Harvard architecture?

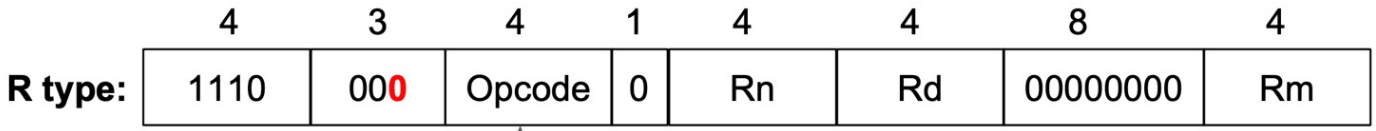
- Harvard Architecture: Use physical **memory interleaving** to achieve the performance increase with having to scale and size two different memory subsystems
- The size of the interleave is some multiple of bytes (like 1024)



x

Assembly & Machine Code Example: ARM-32 (32-bits)

Consider an addition statement
R0 = R1 + R3;



0000 - AND
0001 - EOR
0010 - SUB
0011 - RSB
0100 - ADD
0101 - ADC
0110 - SBC
0111 - RSC
1000 - TST
1001 - TEQ
1010 - CMP
1011 - CMN
1100 - ORR
1101 - MOV
1110 - BIC
1111 - MVN

Simple R-type instructions follow the following template:

```
OP    Rd, Rn, Rm
```

Assembly Language (human readable)

```
ADD   R0, R1, R3
```

machine code pattern in memory

1110 0000 1000 0001 0000 0000 0000 0011

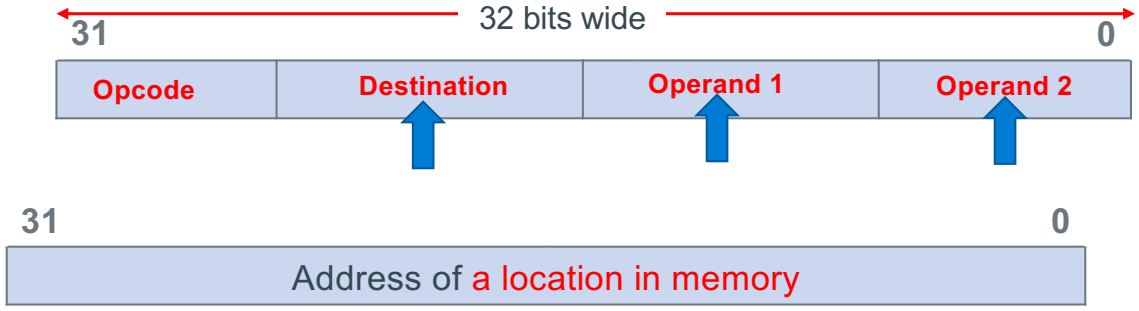
Observe there are only enough bits to specify 16 registers

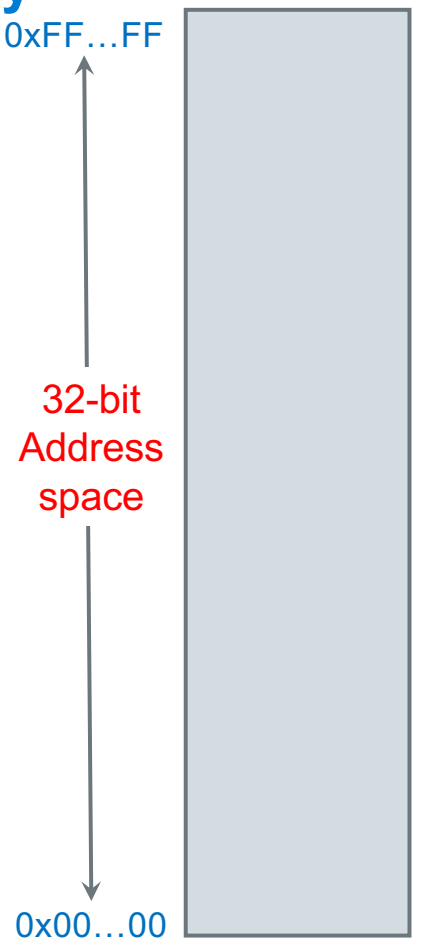
List of Different operations for this type of instruction

x

Why only 16 Registers & how to access all of memory

- Consider a = b + c are operands are in memory
  - Operation code: add — Destination: a
  - Operand 1: b — Operand 2: c
- Aarch32 Instructions were designed to always be: 32 bits wide
  - Some bits must be used to specify the operation code
  - Some bits must be used to specify the destination
  - Some bits must be used to specify the operands
- To address all of memory you must store an address in a register
  - ARM-32 registers (the contents) are 32-bits (can contain data or an address)

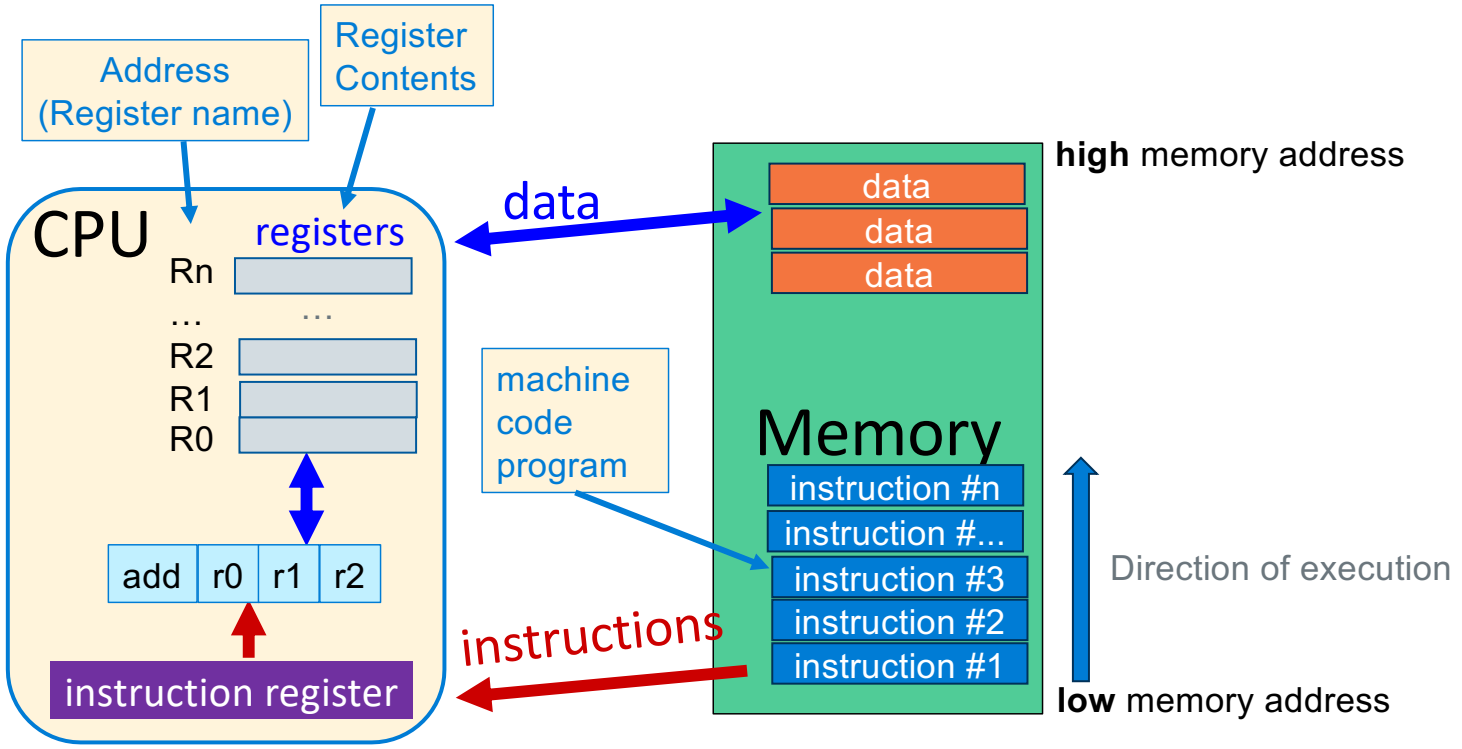




NOT ENOUGH BITS for FULL Addresses to be stored in the instruction

x

Machine code execution order



- **Execution order**: Programs execute from instructions located in **low address** memory to **high address** memory stepping **one machine instruction at a time** (called **execution order**) **unless there is a branch** (example: loop, if statement etc.)

x

From Source code to Execution

```
$ cat test.c
#include <stdlib.h>
#include <stdio.h>
int main (void)
{
    printf("Hello!\n");
    return EXIT_SUCCESS;
}
$ gcc -Wall -Wextra -Werror -c -S test.c
$ ls -ls
total 8
4 -rw-r--r-- 1 kmuller kmuller 109 Mar 14 15:57 test.c
4 -rw-r--r-- 1 kmuller kmuller 725 Mar 14 15:58 test.s
$ gcc test.s
$ ls -ls
total 16
8 -rwxr-xr-x 1 kmuller kmuller 7708 Mar 14 15:58 a.out
4 -rw-r--r-- 1 kmuller kmuller  109 Mar 14 15:57 test.c
4 -rw-r--r-- 1 kmuller kmuller  725 Mar 14 15:58 test.s
$ ./a.out
Hello!
```

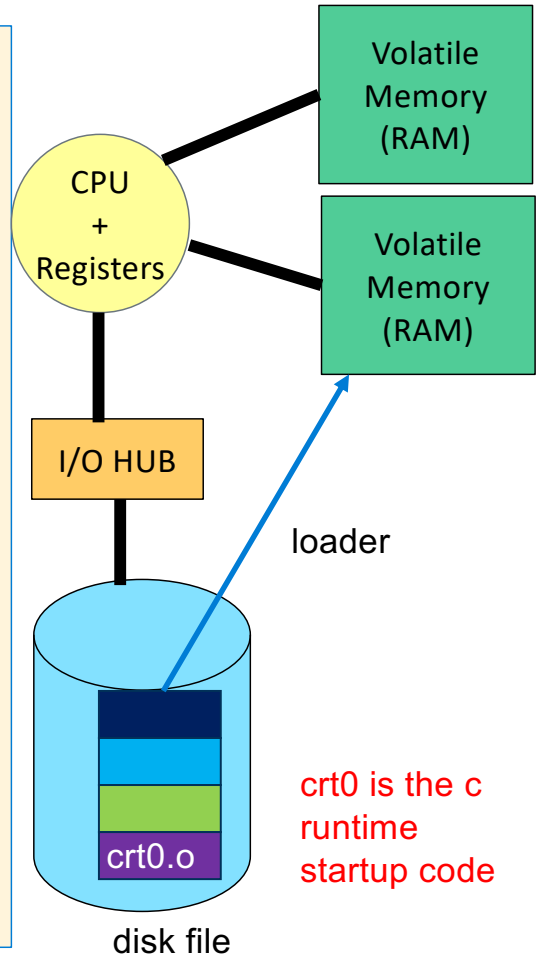
Source to Execution Steps

1. **Compile** (c source to assembler)
2. **Assemble** (assembler source to object)
3. **Link** (Combine object files to executable)
4. **Load** (Copy executable into memory)
5. **Execute** (OS runs the code)

compile

assemble and link
gcc automatically calls the assembler with .S or .s files

load and then execute



x

Equivalent Code: C -> Assembly -> Machine

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    printf("Hello!\n");
    return EXIT_SUCCESS;
}
```

C source

```
        .section .rodata
mesg:   .string "Hello!\n"
        .text
        .global main
        .type   main, %function
        .equ    FP_OFF, 4
        .equ    EXIT_SUCCESS,   0
main:   push    {fp, lr}
        add     fp, sp, FP_OFF
        ldr     r0, L1
        bl      printf
        mov     r0, EXIT_SUCCESS
        sub     sp, fp, FP_OFF
        pop     {fp, lr}
        bx      lr
L1:     .word mesg
```

ARM-32 assembly

address of mesg

Code aka TEXT

memory address | high | low bytes contents | corresponding assembly

```
00010408 <main>:
   10408: e92d4800       push {fp, lr}
   1040c: e28db004       add fp, sp, 4
   10410: e59f0010       ldr r0, [pc, 16] //10428 <L1>
   10414: ebffffb3       bl 102e8 <printf@plt>
   10418: e3a00000       mov r0, 0
   1041c: e24bd004       sub sp, fp, 4
   10420: e8bd4800       pop {fp, lr}
   10424: e12fff1e       bx lr

00010428 <L1>:
   10428: 0001049c

0001049c <mesg>:
   1049c: 6c6c6548       // 'l', 'l', 'e', 'h'
   104a0: 000a216f       // '\0', '\n', '!', 'o'
```

Machine code (instructions)

address of mesg

Data

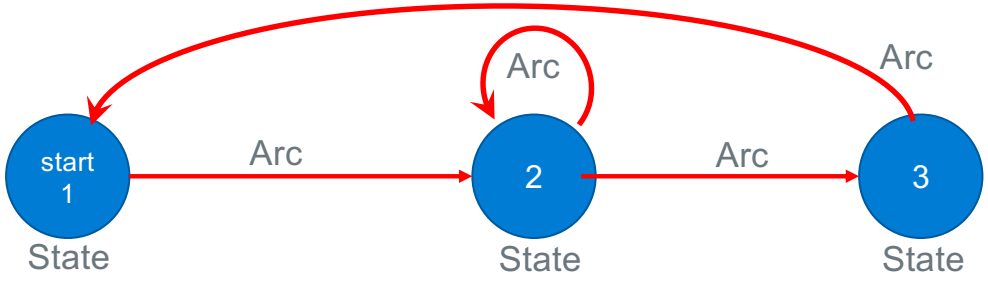
x

PA2/PA3 Design: Using a Finite State Machine

- Finite state machine (or Finite State Automaton) is a way of representing (or *detecting*) a *language*
  - Example: set of string patterns (e.g., ***HA***) *accepted* or *rejected* based on an **input sequence**

Circle (States) and **Arc** representation

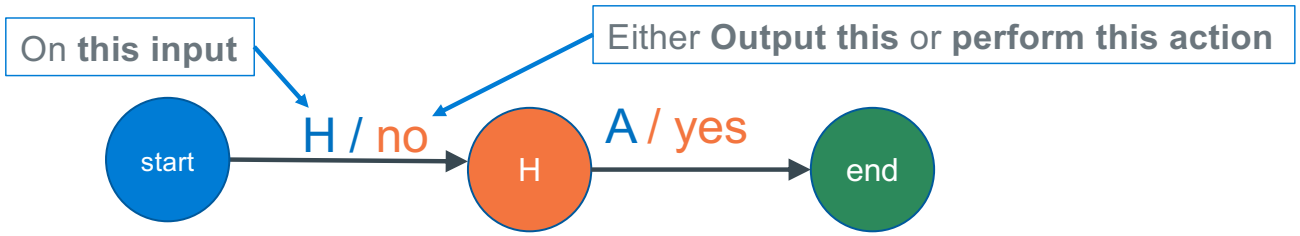
- A **circle** (state) **represents** ***(remembers – a "path")*** **what has already been seen** in the **input stream**
- An **arc** represents a **transition** from one state to the next state for a **specified input** and may **specify** an **optional output** (or **operation to be performed**)
  - The **next state** can be the **same state** or a **different state**
- At any point in time, **one of the states** is the **current state** of the machine
  - **Current state** "remembers" the **input sequence seen so far** by the machine
- **Whenever a state is entered**, it ***"reads"*** to get the next input (except the ***end*** state – next slide)



x

Machine States and Transitions

- Two Special states
  - start `start` state (machine starts "powers up" in this state) **required**
  - end **end** state (done or final state) **not required** – if not present DFA runs forever
- Each **arc** has a label(s) that uses the **notation**: **input1, …, input n / output or action taken**
  - When the **input to the machine matches one of the input labels**, it **selects** that **arc to be taken**
  - The **arc taken** also specifies the output produced or action taken
    - it is **ok to have no output,** or **no operation** associated with an arc
- **Example: FSA machine below** recognizes the sequence **HA** on an input stream, then stops
  - **Question:** what is missing here? – What do we do for inputs NOT specified?



x

Arc labeling

- **output**(c) indicates c is to be output (printed for example)
- An action of – means no action (or output)

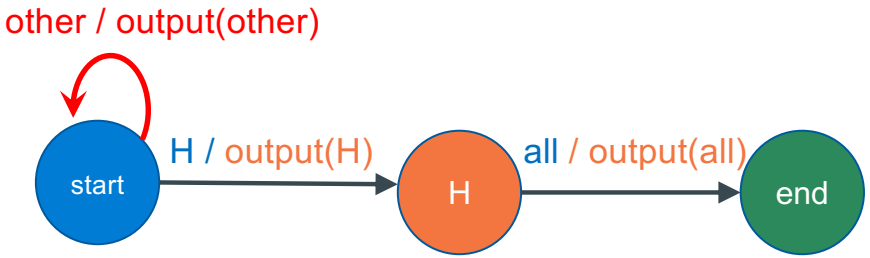
```
a / -
```

- The labels **all** and **other** have special meanings
- When an **arch is labeled** with an input of **other**, this represents all other character inputs that are not specified by other arcs
  - If you need ***to output the actual input character***, you will label the arch as:

```
other / output(other)
```

- When an **arch is labeled** with an input of **all**, then this arc is taken for all inputs
  - this **must be the only arc out of the state**
  - **Question**: Is the all label really needed?
  - If you need to output the actual input character, you will label the arch as:

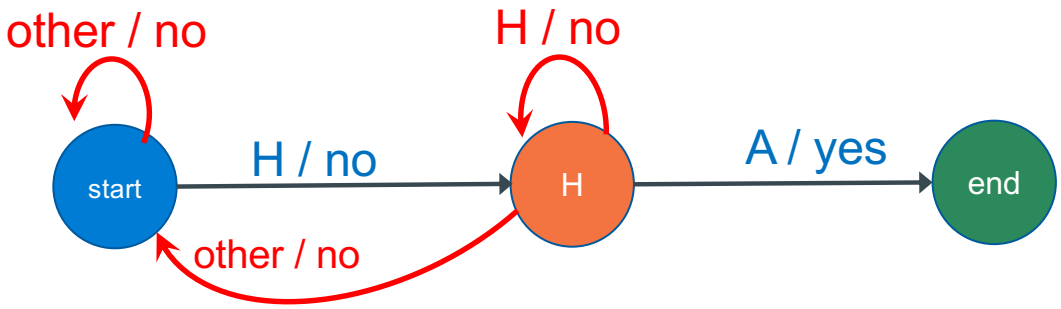
```
all / output(all)
```



x

Designing a Deterministic Finite State Automaton

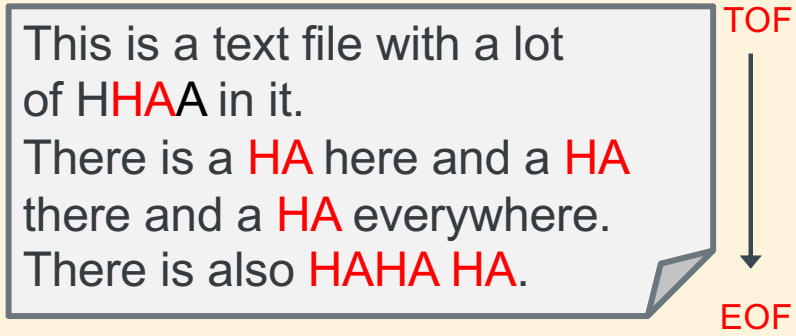
- **Deterministic Finite State Automaton (or deterministic finite state machine)**
  - For any **given state**, then for **all possible inputs**, there is always **one next state**
- Step 1: Define the states (using the recognizer example from the previous slide)
  - ***Start (initial or power up) state:*** input has not seen an H (or no input so far)
  - ***H state***: input has seen at least one H (or more than one H
  - ***end state***: input has seen an H immediately followed by an A
- Step 2: Define the arcs
  - Specify arcs at each state for all possible inputs **(an arc can be taken on more than on input)**
  - Specify output or action (if any) on each arc
  - **Check**: each state transition (arc) is *unambiguous* (unique – a specific input selects just one arc)



x

DFA counting the instances of a pattern

- The state machine on the previous slide would stop after seeing the first HA, and does not take any more input, missing later occurrences of HA in the input
- Say you want to process the entire contents of a text file to find and count all HA's
  - from the top (top of file)
  - to the bottom (end of file)



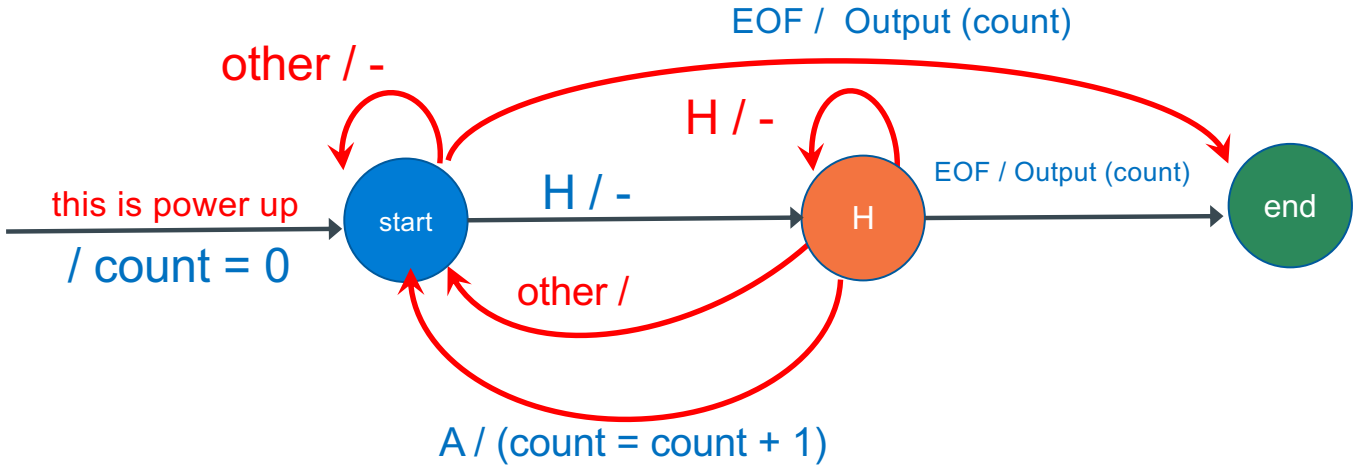
- **Action:** *Alter* the machine to process input from the file in sequential order until the end of the file (EOF) is reached

x

DFA counting the instances of a pattern - 2

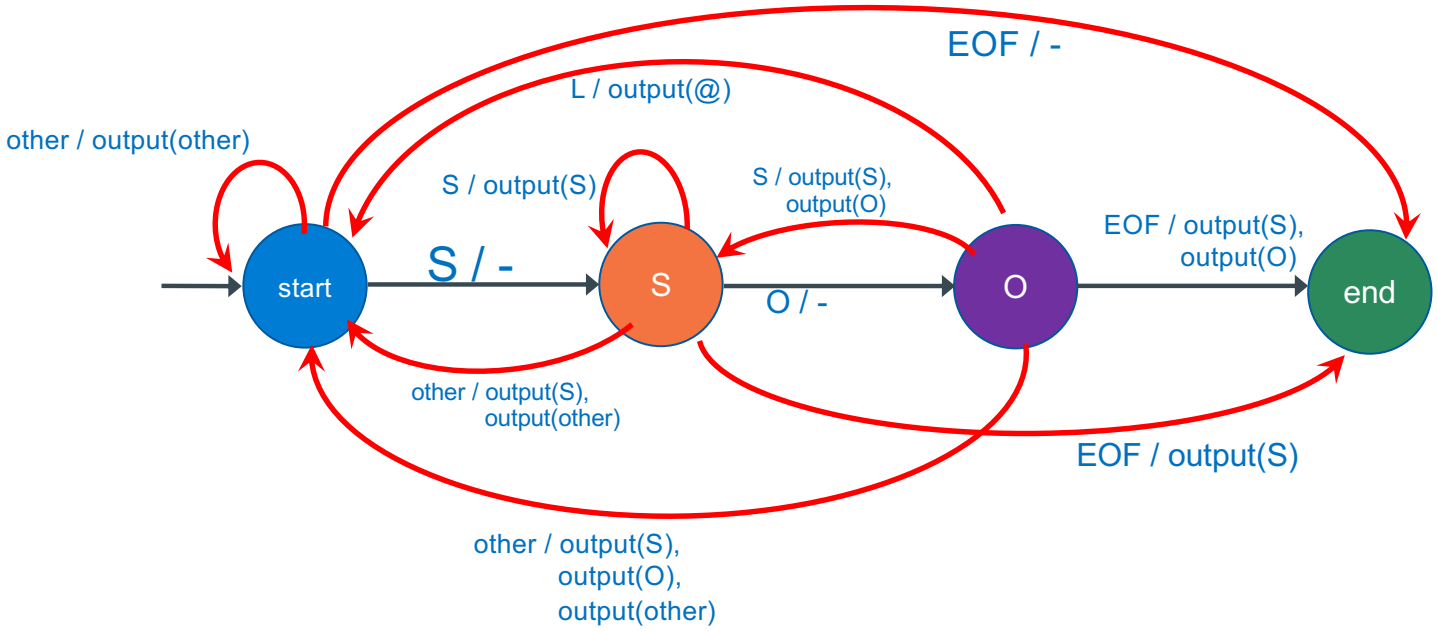
- We will adjust the DFA to act on continuous input (multiple instances of the pattern)
  - And to count the number of HA patterns

1. *"redirect"* the arc(s) that **pointed** at the **end state** to point to the **start state**
2. Convert output to counting actions
3. Add arcs from each state when EOF on input is detected to the eof state



x

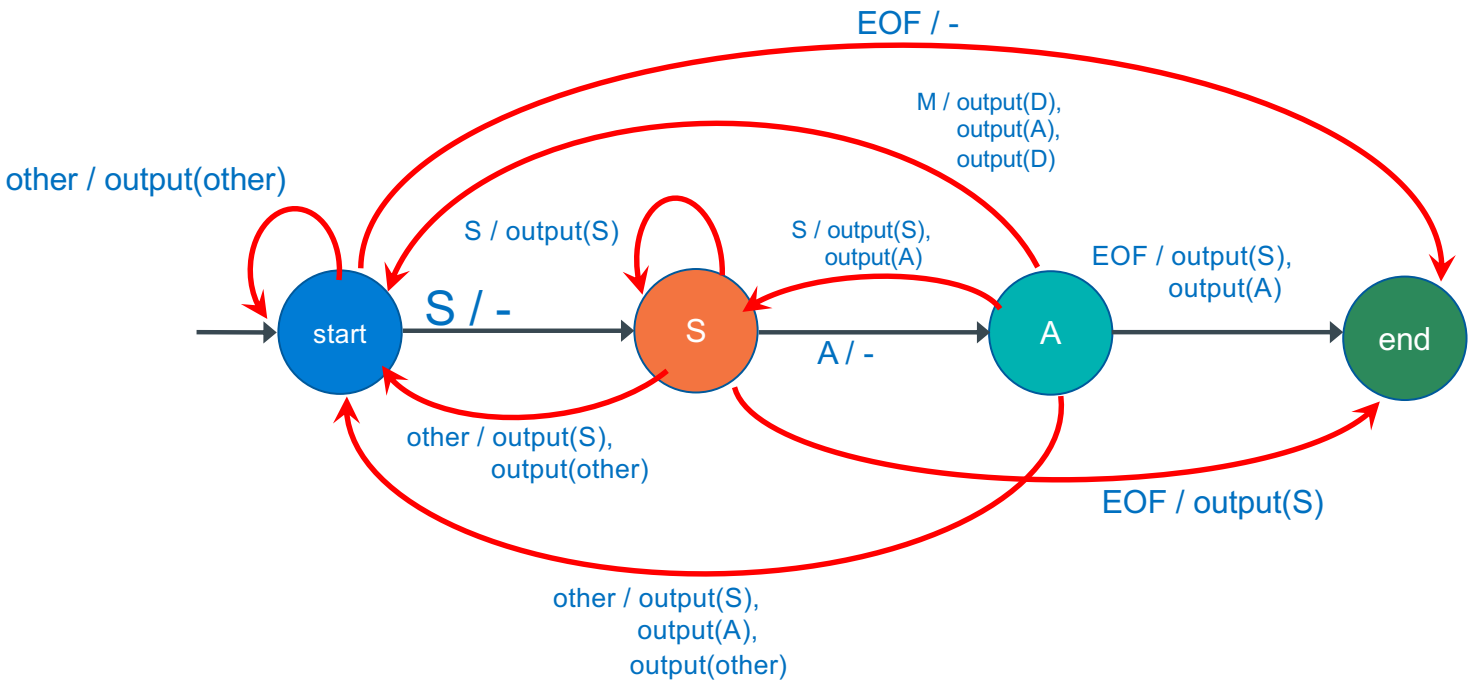
Merging DFA's: Step one design each sequence -1



This DFA replaces SOL with a @

x

Merging DFA's: Step one design each sequence - 2



This DFA replaces SAM with DAD

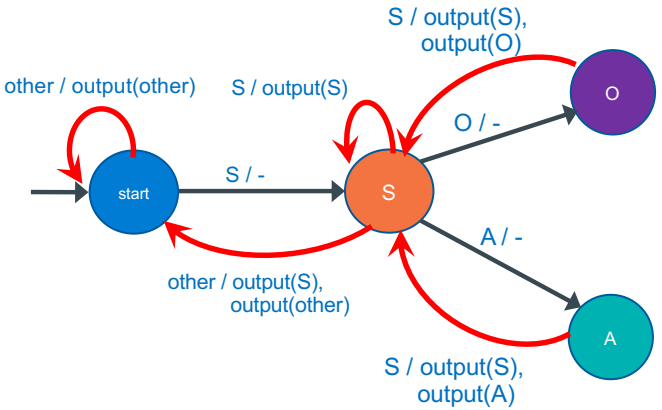
x

Merging DFA's: Step one design each sequence - 3

- To merge two DFA's
  - **combine common states**
  - make sure all the arcs out of the combined states represent the arcs in the two DFA's

Example:

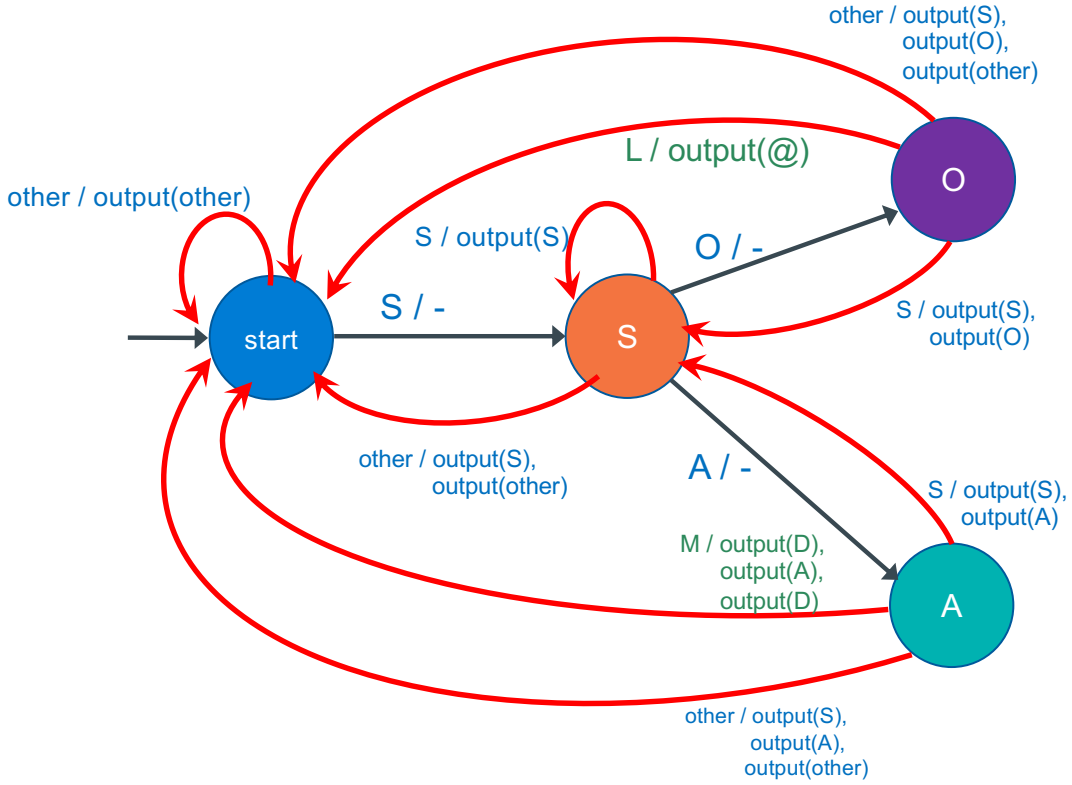
- The prior two DFAs both have the same initial state sequence (**start** and **S**) which will be in the merged DFA
- Next add all the unique arcs out of state S from both the separate DFAs and adjust the labels if necessary
- **Add the remaining states and arcs** that are unique to each DFA
- We are going to **simplify the combination** process by assuming the **input is infinite in length**
  - So, both EOF processing and the end state is not shown
  - **You will use this same assumption in PA2**



x

Merging DFA's – 3 (Finished)

This DFA replaces SOL with a @
and This DFA replaces SAM with DAD



x

Introduction: C Program Structure (Single file)

```c
#include <stdlib.h>
#include <stdio.h>
/*
 * This is a block comment
 */
// this is a line comment

int main(int argc, char *argv[])   // or int main() or int main(void)
{
    char x = '\n';

    printf("Hello World!%c", x);   // "Hello World!\n"
    return EXIT_SUCCESS;           // main always returns either
                                   // EXIT_SUCCESS or EXIT_FAILURE
}
```

directives to the preprocessor

main() is the first function to run
Every executable program must have one function called main()

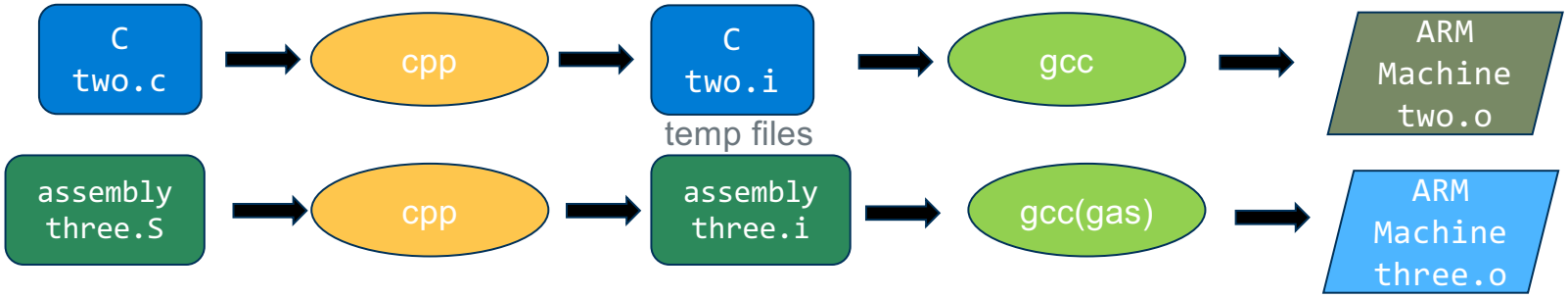
char literal '\n'

library function for writing to stdout

string literal "Hello World!%c"

x

What is the preprocessor (cpp)?



- **Preprocessing is the first phase** in the compilation (.c files) or assembly (.S files only) process
- The **preprocessor** (`cpp`) *transforms* your source code, then **passes it to the compiler** (on .c files) **or the assembler** (on .S files only, not .s files)
  - **cpp is automatically invoked by `gcc`** (when compiling .c or assembling .s files)
- Usually, the input to **cpp** is a C source file (.c) or an assembly source file (.S only) and output from **cpp** is still a C file or assembly file
  - output from cpp is in a temporary .i file (deleted after use)
  - cpp **does not** modify the input source file
- **Common use**: When a **program is divided across multiple source files** (including library files), cpp helps you keep consistency among the files (**one version of the truth**)
  - Examples: Consistent values for a constants, correct function definitions, etc.

x

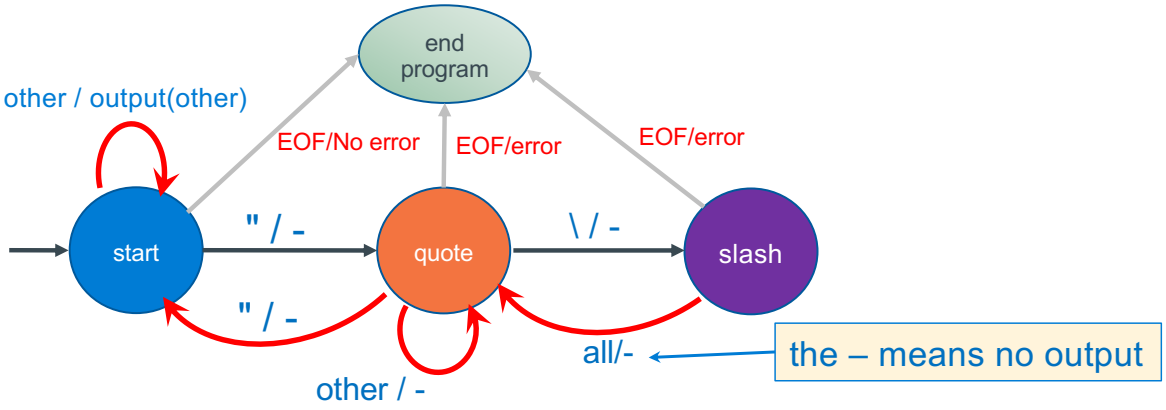
Reference Slides

- Slides in this section are not used in class but contain material that you will find useful

x

PA3: Programming a Deterministic Finite Automaton

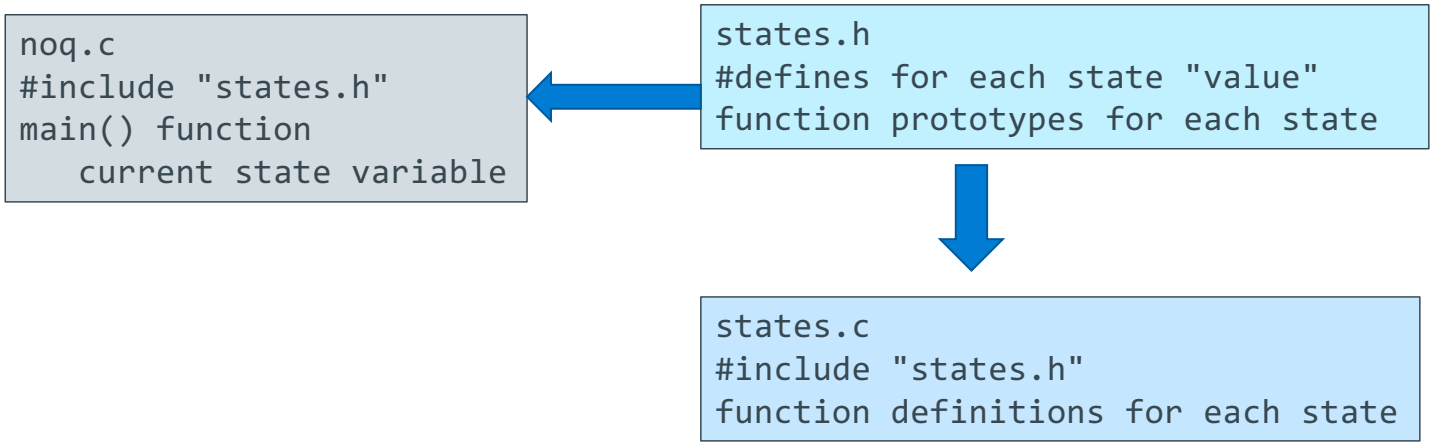
```
Rules for this DFA example
Copy input to output while removing everything in "strings" from output

      input:  ab"foo"cd
      output: abcd

Special Case: If Inside a string, a \ is an escape sequence, ignore the next char
            Allows you to put an " in a string

      input:  ab"foo\"bar"cd
      output: abcd
```



x

Programming a Deterministic Finite Automaton – The Files

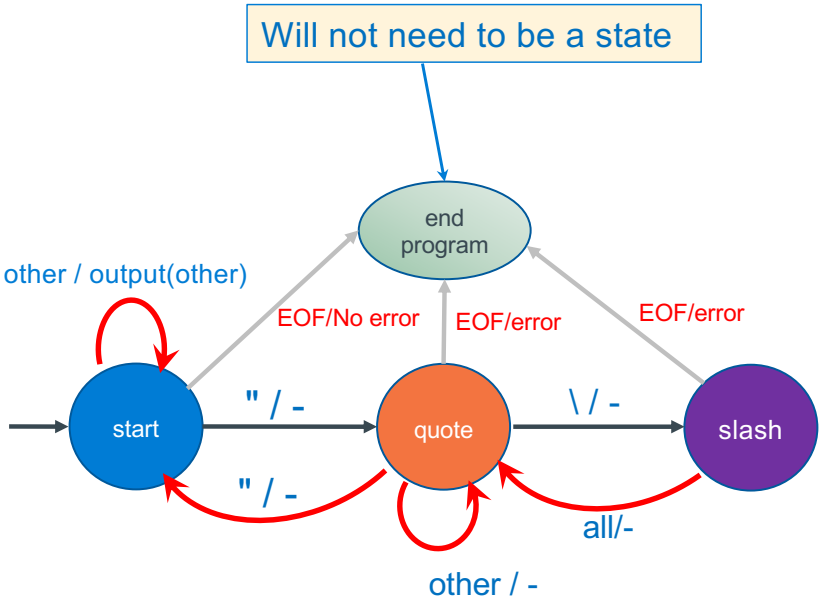
- Break the program into three files
- `noq.c` is where main loop is, imports declarations in `states.h`
- `states.h` is the public interface to the state handlers in `states.c`
- `states.c` definition of the state handler functions, imports declarations in `states.h`
- Observe there is no .h file for noq.c, as it does not have any exports



x

Programming a Deterministic Finite Automaton - states.h

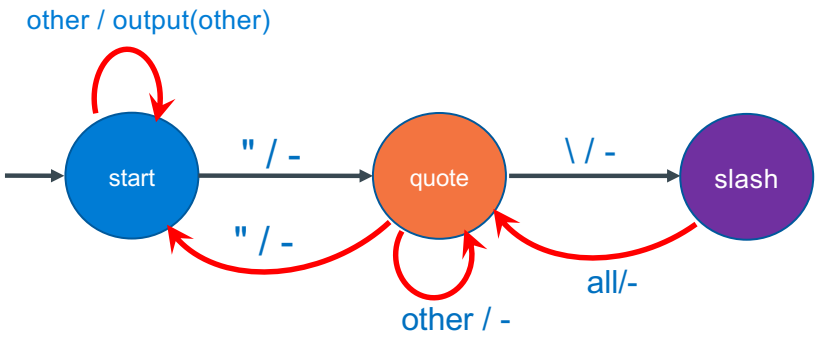
```
// public interface file states.h
#ifndef STATES_H
#define STATES_H

// Assign a value for each state
#define START   0
#define QUOTE   1
#define SLASH   2

// Function prototypes
// for each state handler
int STARTstate(int);
int QUOTEstate(int);
int SLASHstate();

#endif
```



- Each function implements the **arcs** out of that state
  1. **returns the next state** based on the input
  2. **performs** any **actions associated** with **arc taken**

x

Programming a Deterministic Finite Automaton – states.c



states.c

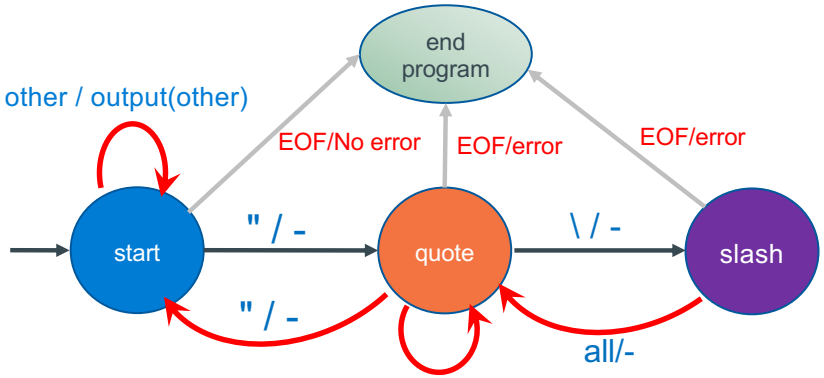
```
#include <stdio.h>
#include "states.h"
int STARTstate(int c)
{
    if (c == '\"')
        return QUOTE;       // saw a double quote
    putchar(c);             // echo input
    return START;           // stay in START
}

int QUOTEstate(int c)
{
    if (c == '\\')
        return SLASH;       // backslash ignore next char
    else if (c == '\"')
        return START;       // closing " go to START
    return QUOTE;
}

int SLASHstate()
{
    return QUOTE;
}
```


x

Programming a Deterministic Finite Automaton – noq.c

primary loop read a char at a time until EOF

process input based on current state




```
int main(void)
{
    int c;                  // input char
    int state = START;      // initial state of DFA

    while ((c = getchar()) != EOF) {
        switch (state) {
        case START:
            state = STARTstate(c);
            break;
        case QUOTE:
            state = QUOTEstate(c);
            break;
        case SLASH:
            state = SLASHstate();
            break;
        default:
            fprintf(stderr, "Error: Invalid state (%d)\n");
            return EXIT_FAILURE;
        }  // end switch
    }  // end while
    /*
     * All done. No explicit end state used here.
     * if not in start state, we have an error
     */
    if (state == START)
        return EXIT_SUCCESS;
    // ok we had an error
    fprintf(stderr, "noq error: Missing end quote \"\n");
    return EXIT_FAILURE;
}
```

call state handlers based on current state
state handlers return next state

check ending "state"

x

Aside: Remember make from CSE15L?

```
# CSE30SP24 DFA Example

# if you type 'make' without arguments, this is the default
PROG        = noq
all:        $(PROG)

# header files and the associated object files
HEAD        =  states.h
SRC         =  noq.c states.c
OBJ         =  ${SRC:%.c=%.o}

# special libraries
LIB         =
LIBFLAGS    = -L ./ $(LIB)

# select the compiler and flags you can over-ride on command line
# e.g., make DEBUG=
CC          = gcc
DEBUG       = -ggdb
CSTD        =
WARN        = -Wall -Wextra -Werror
CDEFS       =
CFLAGS      = -I. $(DEBUG) $(WARN) $(CSTD) $(CDEFS)

$(OBJ):     $(HEAD)

# specify how to compile/assemble the target
$(PROG):    $(OBJ)
    $(CC) $(CFLAGS) $(OBJ) $(LIB) -o $@

# remove binaries
.PHONY: clean clobber
clean:
    rm -f $(OBJ) $(PROG)
```


X

Programming a Deterministic Finite Automaton - testing

```
$ make
gcc -I. -ggdb -Wall -Wextra -Werror     -c -o noq.o noq.c
gcc -I. -ggdb -Wall -Wextra -Werror     -c -o states.o states.c
gcc -I. -ggdb -Wall -Wextra -Werror   noq.o states.o  -o noq
$ ./noq
123"456"789
123789
"123"45"67"
45
"123
456
78"9
9
"test
^d
noq error: Missing end quote "
$ cat in
line1"34"
"line2"line2
line3"
line4
$ ./noq < in > out
noq error: missing end quote "
$ cat out
line1
line2
line3$
```

typed input in red
output in blue

x

C Versus Java

Note: Sorry for the "poor" code indentation; adjusted to fit into the table

	Java	C

Overall Program Structure	source file: Hello.java
	
	public class Hello
	{ public static void main
	(String[] args)
	{ System.out.println(
	"hello world!");
	}
	}
		source file: hello.c
		#include <stdio.h>
		#include <stdlib.h>
		
		int
		main(void)
		{
		printf("hello world!\n");
		return EXIT_SUCCESS;
		}

Access a library	import java.io.File;	#include <stdio.h> // may need to specify library at compile time with -llibname
Building	% javac Hello.java	% gcc –Wall –Wextra -Werror hello.c –o hello
Running (execution)	% java Hello hello world!	% ./hello hello world!

x

C Versus Java

	Java	C

Strings	`String s1 = "Hello";`	`char *s1 = "Hello";  // pointer version`
		`char s1[] = "Hello"; // array version`

String Concatenation	`s1 + s2` `s1 += s2;`	`#include <string.h>` `strcat(s1, s2);`
Logical ops	`&&, ||, !`	`&&, ||, !`
Relational ops	`==, !=, <, >, <=, >=`	`==, !=, <, >, <=, >=`
Arithmetic ops	`+, -, *, /, %, unary -`	`+, -, *, /, %, unary -`
Bitwise ops	`<<, >>, >>>, &, ^, |, ~`	`<<, >>, &, ^, |, ~`
Assignment ops	`=, +=, -=, *=, /=, %=,` `<<=, >>=, >>>=, &=, ^=, |=`	`=, +=, -=, *=, /=, %=,` `<<=, >>=, &=, ^=, |=`

x

C Versus Java

	Java	C

Arrays	`int [] a = new int [10];`
	`float [][] b =`
	`  new float [5][20];`
		`int a[10];`
		`float b[5][20];`

Array bounds checking	`// run time checking`	`// no run time checks – speed optimized`
Pointer type	`// Object reference is an` `// implicit pointer`	`int *p;` `char *p;`

Record type	`class Mine`
	`{  int x;`
	`  float y;`
	`}`
		`struct Mine`
		`{  int x;`
		`  float y;`
		`};`

x

C Versus Java

	Java	C
if, switch, for, do-while, while, continue, break, return	`// equivalent`	`// equivalent`
exceptions	`throw, try-catch-finally`	`// no equivalent`
labeled break	`break somelabel;`	`// no equivalent`
labeled continue	`continue somelabel;`	`// no equivalent`

calls:
Java method
C function
	`f(x, y, z);`
	`someObject.f(x, y, z);`
	`SomeClass.f(x, y, z);`
		`f(x, y, z); // other differences, later…`

x

C Programming Toolchain - Basic Tools

- **gcc**
  - Is a front end for all the tools and by default will turn C source or assembly source into executable programs
- **preprocessor**
  - Insertion into source files during compilation or assembly of files containing macros (expanded), declarations etc.
- **compiler**
  - Translates C programs into hardware dependent assembly language text files
- **assembler**
  - Converts hardware dependent assembly language source files into machine code object files
- **Linker (or link editor)**
  - Combines (links) one or more object files and libraries into executable program files
  - this may include modification of the code to resolve uses with definitions and relocate addresses

x

C Programming Toolchain: The Source files

- The C development toolchain uses several different file types (indicated by .suffix in the filename)
- filename.**h** public interface *"header or include files" often used as <filename.h> or "filename.h"*
  - common contents: public (exported) function and variable declarations, and constants and language macros
  - Processed by **cpp** (the C pre-processor) to do inline expansion of the include file contents and insert it into a source file before the compilation starts, enables consistency
- filename.**c**
  - a source text file in C language source
  - Processed by **gcc**
- filename.**S**
  - a source text file in hardware specific assembly language (programmer created)
  - processed by gcc which calls gas (assembler)
- filename.**s**
  - machine generated by the compiler from a **.c** file
  - processed by gcc which calls gas (assembler)

x

C Programming Toolchain: The Generated files

- filename.**o** *"relocatable object file"*
  - Compiled from a single source file in a .c file or assembled from a single .s file into machine code
  - A **.o** file is an incomplete program (not all references to functions or variables are defined) this code will not execute
  - The **.o** and **.c, .s,** or **.S** files share the same root name by convention
  - created by gcc calling ld (linkage editor)
- library.**a** *"static library file"*
  - aggregation of individual **.o** files where each can be extracted independently
  - during the process of combining **.o** files into an executable by the linkage editor, the files are extracted as needed to resolve missing definitions
  - created by **ar**, processed by **ld** (usually invoked via **gcc**)
- a.out *"executable program"*
  - Executable program (may be a combination of one or more **.o files and .a** files) that was compiled or assembled into machine code and all variables and functions are defined
  - processed by **ld** (usually invoked via **gcc**)

x

Basic gcc toolchain usage

- Run gcc with flags
  - **-Wall -Wextra**
    - required flag for c programs in cse30
    - output all warning messages
  - **-c**
    - Optional flag (lower case)
    - Compile or assemble to object file only do not call **ld** to link
    - creates a **.o** file
  - **-ggdb**
    - *Optional* flag
    - Compile with debug support (gdb)
    - generates code that is easier to debug
    - removes many optimizations
  - **-o** *<filename>*
    - specifies *filename* of executable file
    - **a.out** is the default
  - **-S**
    - *Optional* flag (upper case S)
    - Compiles to assembly text file and stops
    - creates a **.s** file

- Producing an executable file
  - **gcc –Wall –Wextra -Werror mysrc.c**
    - creates an executable file ***a.out***
- To use a specific version of C use of one the std= option
  - **gcc –Wall –Wextra –Werror –std=c11 mysrc.c**
- Producing an object file with gdb debug support add -ggdb
  - **gcc –Wall -Wextra –Werror –c –ggdb mysrc.c**
    - creates an object file **mysrc.o**
  - **gcc –Wall –Wextra -Werror –c –ggdb mymain.c**
    - creates an object file **mymain.o**
- Linkage step
  - combining a program spread across multiple files
  - **gcc –Wall -Wextra –Werror –o myprog mymain.o mysrc.o**
    - creates executable file **myprog**
- Compile and linkage of file(s) in one step
  - **gcc –Wall –Wextra -Werror -o myprog mysrc.c mymain.c**
- run the program (refer to cse15l notes)
  - **% ./myprog**