

Introduction and Course Basics

September, 2011
Autumn Quarter

CSE 560N

1

Welcome to 560

- Graders
 - Josh Meeks
 - Jenna McAuley
- Contacting us
 - al@cse.ohio-state.edu, w: 247-7338 h:876-4206 c: 561-1823
- Office Hours Dreese 297
 - After class and as scheduled
 - Grader hours to be determined

CSE 560N

2

Grading and Team Projects

- Exams: midterm and final
- SP: Software Projects
- PE: Peer Evaluations
- Penalties
- Real time grading and presentations
- 65% Rule to pass
 - You must get an average of at least 65% on the exams
 - You must get an average of at least 65% on the software projects
 - You must get an average of at least 65% on the Peer Evaluations

CSE 560N

3

Skills needed/learned

- HEX
- Hand assembly
- Team survival and success
- Complete System implementation
- Software Engineering techniques
 - Design
 - Coding structure and standards
 - Code management and change management
 - Testing and validation
 - Presentation skills
- Basic systems concepts
- Improve writing, organizational, and presentation skills

CSE 560N

4

Languages/Hardware

- C++ Visual Studio .NET 2003, C#, JAVA, or ???

Make sure everyone in the group uses the same Hardware, compiler, and operating system.

Integration of the code should be on going...
Don't wait until close to the due date to bring the Code together.

Note: You should be aware that many of the routines you wrote for other CSE classes can be used. For example you have written a parser, developed BNF techniques, in-memory tables, etc.

CSE 560N

5

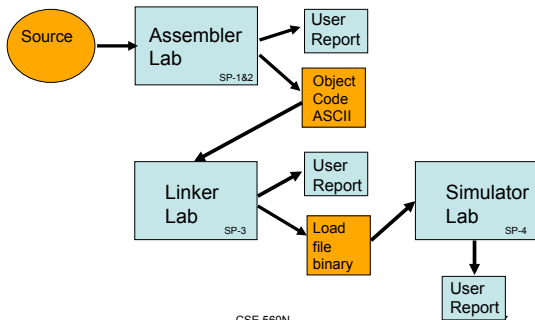
Groups

- Pick your own group TODAY
 - groups 5 **Maximum!** 4 **Minimum!**
 - full time students vs. part time
- Team conflicts
 - Resolution - How will you do this?
- Equal work
 - Differential grading
 - All phases-Design, coding, testing, documentation
- Communicate
 - Have regular team meetings - keep minutes
 - Clear assignments and agreements

CSE 560N

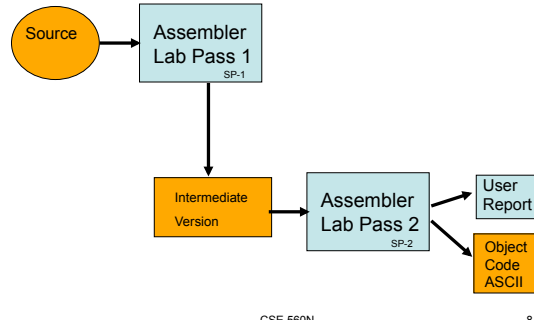
6

Lab Overview



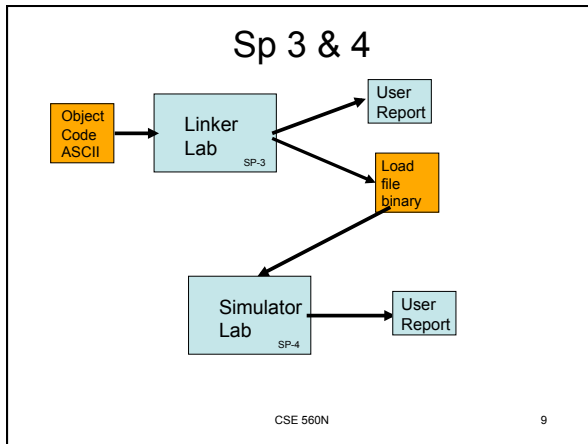
CSE 560N

SP1&2



CSE 560N

8



Documentation

- Software Guide
- Developer Guide
- DED
- Test Plan
- Creativity
- Writing Quality

Hint: Do not simply plagiarize the course handouts
Hint: Write your Documentation BEFORE you code

CSE 560N 10

Test Plan

- Planned
- Logical
- Consistent - all the same source
- Meaningful
- Test limits
- Hints:
 - Design for testing as you design the code
 - Test modules as you write them
 - Summarize purpose of each test
 - Have a test Table of Contents

CSE 560N 11

Log Book & Errata Report

- Log Book
 - Record of meeting meetings (notes-minutes)
 - Record of team member commitments
- Errata Report
 - List of known errors
 - List of known omissions
 - Must be provided or there is 10 point penalty

CSE 560N 12

Software Engineering

- Planning
- Structure
- Specifications are 95% accurate
- Decision Table
- Error Table

CSE 560N

13

Software Engineering

- Use a modular design you will use many routines across all labs
- Write a single routine for table searching and updating across all tables
- Use built in functions
- Divide the work evenly across all areas of the assignment
 - Everyone writes code
 - Everyone documents
 - Everyone designs
 - Everyone tests

CSE 560N

14

Software Engineering

- Use a top down approach for design
- Draw a module function diagram
 - show variables needed for the module and results returned

Note: If you are familiar with UML consider using it.

CSE 560N

15

Software Engineering-Testing

- Test the modules independently
- Test the modules as they are added to the system
- Develop a realistic set of test programs that are run after each change. Think about a regression testing approach
- Test data in and parameters passed
- Don't be a run and gun programmer

Note: In early CSE classes you are taught that information passed to you should be trusted. In this class I encourage you to not blindly trust passed data. Always test passed parameters to verify they are within the expected range.

CSE 560N

16

Software Engineering-Testing

- Levels of testing
 - Unit isolation testing
 - Black box -- blind tests
 - White Box -- tests based on module knowledge
 - Integration Testing
 - Bottom up testing
 - Top down testing

Note: Consider allowing the end-user to enter a code on the command line that would turn debugging on or off. This will be a useful tool for your team also. AS you are testing it will be nice to see intermediate results and status, but for your final runs the debug feature should be turned off.

CSE 560N

17

Software Engineering Leadership

- Identify key leaders for primary functions
 - Design
 - Documentation
 - Coding
 - Testing
 - Project manager

Note: Take your role as a leader very seriously and keep your team-mates on track. The overall project manager needs to be kept informed of progress and problem areas. The leaders have an obligation to report to the instructor when the team is missing deadlines. This is especially true if a team-mate is not participating.

CSE 560N

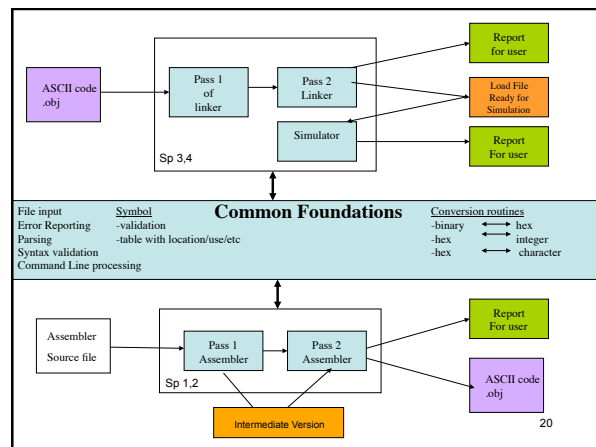
18

Software Engineering

- Design for upgrades and changes
- Design with common routines (re-utilizable modules)

CSE 560N

19



20

Software Engineering

CSE 560N

21

Team issues

- What if I feel that my teammates are not doing their "fair" share?
- One teammate is a coding "whiz-kid" and has decided to simply do it all.
- What if two of your teammates are long time buddies and they do everything together (software wise) and leave you out.
- What is differential grading? How can we avoid it?
- Can I still pass the class without doing anything on the labs?

CSE 560N

22

Coding rules/standards

- 1. Meet or Surpass company-coding standards
- 2. Should be CLEAR STRAIGHT forward coding
- 3. Should use library functions (don't re-invent SQRT, SIN or any company-supplied functions.)
- 4. Avoid lots of temporary variables
- 5. Let the system do some of the dirty work i.e. bit conversion don't write your own.
- 6. Replace repetitive expressions with a function or subroutine.
- 7. Use meaningful variable names—follow company standards.
- 8. Use parenthesis to avoid confusion
- 9. Use arrays to avoid repetitive sequences.
- 10. Modularize
- 11. If doing maintenance: If the code is bad rewrite don't patch it.
- 12. Use recursive procedures when and only when necessary
Test the program module by module FIRST.
- 13. Give meaningful error messages recover where necessary.
Tell what the error is and how to fix it.
- 14. Don't just find one error and quit -- continue on and find all the errors.

CSE 560N

23

Coding Rules/Standards

- 15. Input should be easy to enter.
- 16. Output should be self-explanatory
- 17. Never trust any information provided by previous developers.
- 18. Don't run and gun

Efficiency

- 19. Make it correct before faster
- 20. Make it fail safe before faster.
- 21. Make the code clear and clean before faster.
- 22. Let the compiler do optimization.

Code Level Documentation

- 23. Make comments and code agree.
- 24. Don't just echo the code in comments. e.g. ADD 3,AB Add AB to register 3
- 25. Don't comment bad code rewrite it.
- 26. Use meaningful statement labels where possible.
- 27. Use meaningful procedure names.

CSE 560N

24

End User Error Messages

- Complete Story
- Don't stop and end on individual messages
- Error message levels
 - Informative
 - Minor
 - Severe
 - Fatal

CSE 560N

25

TESTING the PROGRAM

- Develop a Test Plan
 - Logical
 - Areas
 - Limits
 - Syntax versus usage
- Prepare your own test data and figure out the expected results by hand.
- Ask someone else to prepare a test for and figure out the expected results by hand.
- Be sure to test all boundaries and input data.

CSE 560N

26

Good Test Plan Contains

- Table of Contents
- For each test
 - Purpose of test
 - Expected results
 - Section of code tree being tested
 - Actual input file
 - All results file

CSE 560N

27

Preparing and testing

- Develop functional tests as code is being developed
- Once tested but before add to source tree have some one else in the group review the code and the test results

CSE 560N

28

The test.

- 1. Use realistic test data
- 2. Test both extremes
- 3. Test each function
- 4. Test each error message

CSE 560N

29

Basic Stages in Software Design

- Requirements Analysis
 - answers: “*What* does the system do?”
 - *understand* the problem
 - deliverables:
 1. requirements document
 - from the user’s point of view
 - *defines* and *limits* the scope of the system
 2. specification document
 - from the developer’s point of view
 - basis for design / implementation / testing

CSE 560N

30

Basic Stages (II)

- High Level Design
 - *identify* and *evaluate* possible solutions
 - simplicity, effort, cost
 - refine the design

CSE 560N

31

Basic Stages (III)

- Design
 - answers: “*How* does the system do what it does?”
 - high-level description of components, interfaces, and interactions
 - abstraction is critical
 - given in terms of data structures, procedures, algorithms, ...

CSE 560N

32

Basic Stages (IV)

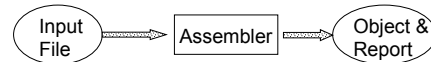
- Module Specification
 - *identify* the abstractions
 - *describe* the abstractions
 - see the specification skeleton in the syllabus
 - describe the interactions (interfaces)
- Two common and broad classes of design:
 1. procedural
 2. object-oriented

CSE 560N

33

Procedural Design

- Focus on the *functionality*
- Create a data-flow view of computation

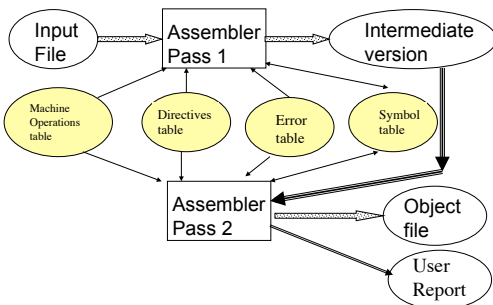


- Use this data-flow to decompose a large system into smaller modules

CSE 560N

34

Procedural Design

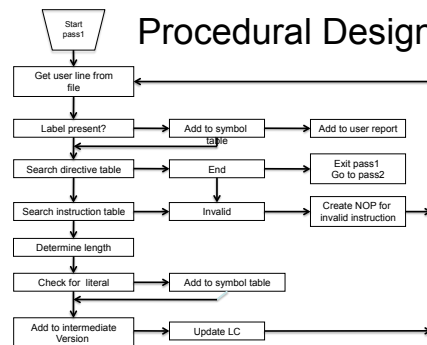


CSE 560N

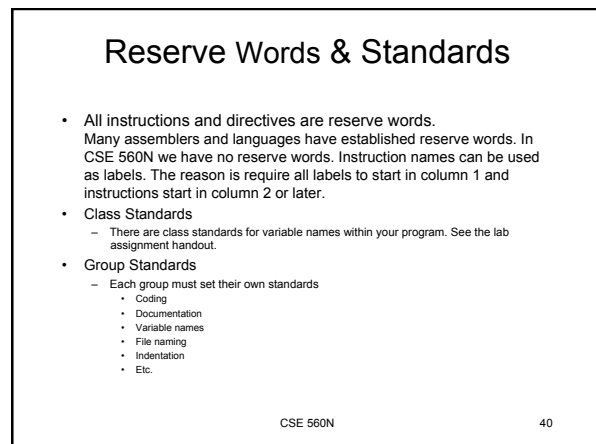
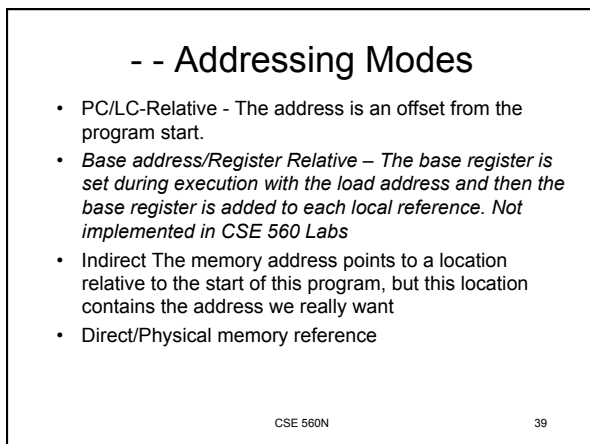
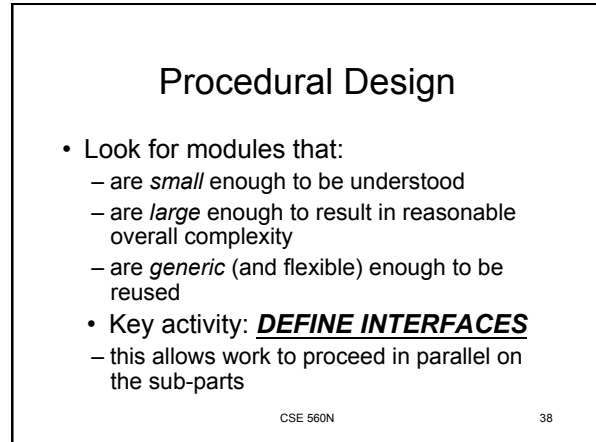
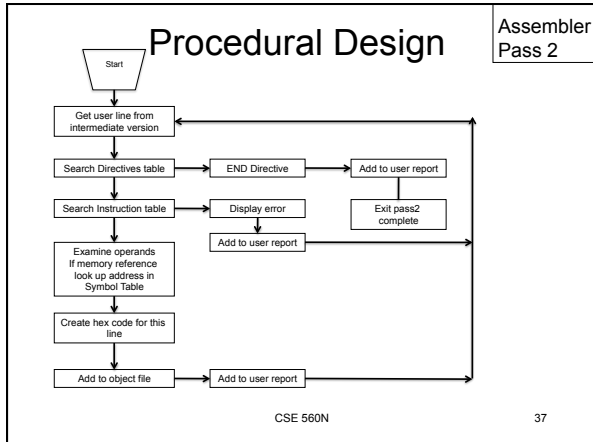
35

Procedural Design

Assembler Pass 1



36



Public/Ext directive-ops

- ENTRy symbol
 - Public Definition -
 - name which must appear as a label somewhere in this program. This symbol may be used as an operand by other programs. Causes a L record to be generated in the object file for each symbol.
- EXTERNAL symbol
 - External Reference
 - declares the symbol r1 to be an external name which must not appear as a label in this program. This symbol may be used as an operand by this program. The symbol must be defined in some other program by an ENTRy. (Loader will watch for this).

CSE 560N

41

Number and Character directive-ops

- num l=nnnnnn
 - Signed integer -2^{31} to $+2^{31}-1$
Two's compliment
- chc c='cccc'
 - ' ' Character string up to 2 characters
must be in single quotes (left justified fill with blank)
watch out for blanks ascii 20
 - Will always set relocation type in object file to ABSOLUTE for these two directives

CSE 560N

42

Assembler Output

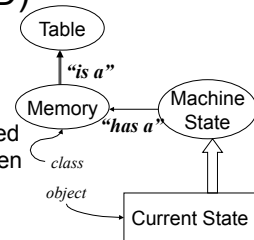
- Assembly Listing - Report to User
 - Complete copy of the users source line
 - Any errors to be reported appear immediately after the line in question
- Object file - Information for the LINKER
- Symbol Table - List of all variables defined or referenced in the program

CSE 560N

43

Object-Oriented Design (OOD)

- Focus on the *data*
- Program = collection of interacting objects
- Sketch out the *types* needed and the *interactions* between these types



CSE 560N

44

OOD: Finding Classes

- Design is often based on reality
- Talk to field experts to understand the system being modeled in software.
- Write down scenarios
 - “use case” analysis
- Draw lots of pictures and refine model
- Decide on class invariants

CSE 560N

45

OOD: Specify Relationships

- Typical class relationships include:
 - inheritance (e.g. “a car *is* a vehicle”)
 - containment (e.g. “a car *has* a steering wheel”)
 - use (e.g. “a car *uses* a highway”)
- Determine responsibility of each class
 - delegate *where appropriate* (not too much!)
- Must strike a balance (small vs large) in the *size* and *functionality* of classes

CSE 560N

46

OOD: Specify Operations

- Important categories of operations:
 - construct, initialize, copy, assign, destroy
 - access, update, iterate
- Set should be small and independent
 - do not implement every possible use / extension
- Focus on behavior, not implementation
 - confirm invariants
 - Key activity: **DEFINE INTERFACES**

CSE 560N

47

Assignments

- Design leader
- Documentation leader
- Testing leader
- Coding leader
- Project leader

CSE 560N

48

Coding Standards

- Global variables in ALL_CAPS with underscores
- Local variables lower_case with underscores
- Function names must be descriptive but short, First letter of each new word is capitalized
- No underscores
- Write preamble before actually coding anything
- Write Documentation before actually coding anything
- Write Unit testing as writing the code

CSE 560N

49

Testing Standards

- Compile a list of errors before testing
- Implement error checking into code
- Test class functions together, test all dependant classes last

CSE 560N

50

Error Handling

- All error handling shall be done with the class ErrorMessagePrinter
- EMP will take errors and store them into a array of errors. At the end of each line all errors from that line will be printed (Error #, Error Message)

CSE 560N

51

Token Types

- MNEMONIC
- DATA
- IMMEDIATE
- SYMBOL
- COMMA
- WHITESPACE
- COMMENT
- ERROR

CSE 560N

52

ParseDataSection

- This function will be using a loop that terminates once it reaches the .Code directive. It will read in each symbol name, its type and its value and store in the symbol table. After each iteration of the loop, the Location Counter will be updated based on the size of the data. The data section will have its own BNF which will be used for checking for errors

CSE 560N

53

.Data BNF

- Terminal:
 - <symbol>, <whitespace>, <type>, <value>, <comma>, <comment>
- Non-terminal:
 - <statement>, <more values>

CSE 560N

54

ParseSymbol

- Function parses the symbol and adds it to the symbol class, value and type will be added to the class variable with ParseType and ParseValue.

CSE 560N

55

Rules for Instruction BNF

- § <operation> -> <symbol><whitespace><mnemonic><rest> | <whitespace><mnemonic><rest> | <mnemonic><rest>
- § <rest> -> <whitespace><rest after ws> | ^
- § <rest after ws> -> <arg><other args> | <comment> | ^
- § <other args> -> <comma><whitespace><arg><other args> | <comma><arg><other args> | <whitespace><comment> | ^
- § <arg> -> <dword ptr> | <byte ptr> | <reg> | <imm> | <symbol>

CSE 560N

56

Main Program Overview

- Load file - "InstructionTable.txt"
- 6 global variables
- 1. INSTRUCTION_TABLE
- 2. LINE_NUM
- 3. LOCATION_COUNTER
- 4. SYMBOL_TABLE
- 5. ASM_SOURCE
- 6. INTERMEDIATE_FILE

CSE 560N

57

INSTRUCTION_TABLE

- Of type InstructionTable
- Contains arrays of each type of instruction
- Public functions:
 - LoadInstructionTable
 - FindAndPrintMatchingInstruction

CSE 560N

58

LoadInstructionTable

- This function opens up "InstructionTable.txt" and saves the instruction information to class objects inside of INSTRUCTION_TABLE. There are five types/formats of instructions found in this file, as defined in a later section.

CSE 560N

59

FindAndPrintMatchingInstruction

- In a nutshell, this function takes many arguments, including all the important fields of a parsed instruction. It goes through all of the private functionality of the InstructionTable class to verify that the instruction is a valid one listed in the table.
 - If so, it prints out as much hex as possible for the current line in the INTERMEDIATE_FILE.
 - If not, it adds the appropriate error codes to the ErrorMessagePrinter.
- Call:
- **void FindAndPrintMatchingInstruction(string mnemonic, Token arg1, Token arg2, ErrorMessagePrinter& error_printer, SymbolTable& SYMBOL_TABLE, int LOCATION_COUNTER, FILE * INTERMEDIATE_FILE):**

CSE 560N

60

LOCATION_COUNTER

- Integer
- Stores the current location

CSE 560N

61

SYMBOL_TABLE

- Variable type: SymbolTable
- Keeps track of symbols being used in the program
- When new label is discovered
 - Validate Syntactically
 - Check if already in the table
 - Store in table with assigned Location counter
- When searching for label
 - Retrieve symbol and assigned location

CSE 560N

62

ASM_SOURCE

- FILE * object
- Points to the .asm source file that is loaded with the program

CSE 560N

63

INTERMEDIATE_FILE

- FILE *
- Points to the output file, "IntermediateFile.txt"

CSE 560N

64

Intermediate file Format

Suggested:

- | | |
|--|----------------------|
| 1. Source line as provided by the programmer | 1. Pass_1_results |
| 1. Parsed Line | 2. opcode binary |
| 2. Assigned LC | 2. IR_Flag binary |
| 2. Label | 2. BR_Flag binary |
| 2. Opcode | 2. Debug_Flag binary |
| 2. Directive | 2. ADR_Flag binary |
| 2. Instruction Operands | |
| 3. op.1 | |
| 3. op.2 | |
| 2. Directive operands | |
| 3. op.1 | |
| 2. Error conditions | |
| 3. Err1 | |
| 3. Err2 | |
| 3. Err.3 | |

CSE 560N

65

Class SymbolTable

Private:

- Symbol table[]

Public:

- SymbolTable();
- ~SymbolTable();
- void DefineSymbol (Symbol symb)
- bool SymbolsDefined (string label)
- void UpdateLocation (string label, int location)
- void UpdateUsage(string label, int usage)
- void GetSymbol (string label, Symbol symb)
- int GetLocation (string label)
- int GetLength(string label)

CSE 560N

66

Class Symbol

Private:

- String label
- int location
- int usage

Public:

- Symbol (string label, int location, int usage);
- Symbol() // default constructor
- ~Symbol() // default destructor
- void SetLabel (string name);
- void SetLocation (int loc);
- void SetUsage (int use);
- void SetLength();
- void GetSymbolFields (string name, int loc, int use);
- string GetLabel ();
- int GetLocation();
- int GetUsage();
- int GetLength();

CSE 560N

67

Instruction Table Organization

- InstructionTypeA first, InstructionTypeB, InstructionTypeC, InstructionTypeD, InstructionTypeE
- Each line terminated with '\n'
- A line with only '\n' signifies end of a particular type

March 28, 2011

CSE 560N

68

Instruction Table Initialization

- Instructions are loaded into according to instruction type until EOF

CSE 560N

69

Main Function

- Initialize INSTRUCTION_TABLE
 - INSTRUCTION_TABLE.LoadInstructionTable()
- Open .asm file, ASM_SOURCE
- Open blank output file, INTERMEDIATE_FILE
 - Intermediatefile.txt
- ParseDataSection()
- ParseCodeSection()

CSE 560N

70

ParseDataSection()

- Parses Line by Line
- Add each symbol to the SYMBOL_TABLE
 - Includes
 - Symbol name
 - Symbol value
 - length
 - Symbol Type

CSE 560N

71

ParseOperation() cont.

- Reads the first Token
 - Label
 - SYMBOL_TABLE.SymbolsDefined(Token) - discover if symbol exists.
 - Defined: adds an error to the error_printer
 - Not-defined: creates a new symbol object
 - Whitespace
 - Skips ahead to find the first token

CSE 560N

72

CodeParser() cont.

- First token is:
 - Mnemonic, saves the text field of the token to the mnemonic variable
 - If the type field of the first is anything else, add an error to error_printer

CSE 560N

73

CodeParser() cont.

- Creates a boolean and set it equal to ParseRest(tokenizer, arg1, arg2, error_printer)
 - If it returns false, add an error code to the error_printer saying it cannot determine which instruction format the line is in.
 - Otherwise calls INSTRUCTION_TABLE.FindAndPrintMatchingInstruction(<params>)
 - Lastly, calls error_printer.PrintErrorReport (INTERMEDIATE_FILE) to print out all of the errors this line produced to the intermediate file.

CSE 560N

74

ParseRest

- Gets next Token out of tokenizer
 - If none, returns true
 - If WHITESPACE token, returns ParseRestAfterWS(tokenizer, arg1, arg2, error_printer)
 - If not a WHITESPACE token, adds an error code to the error_printer and returns false.

CSE 560N

75

ParseRestAfterWS

- This function creates a boolean variable parse_arg_succeeded, then gets the next token out of tokenizer (the passed TokenizingMachine), if there is one
 - If there isn't and tokenizer is empty, the function just returns true.
 - If there is a token and it is a COMMENT token, it tries to get the next token after it.
 - If there is one, then it adds the error code to the error_printer about expecting nothing on a line after a comment. (This really shouldn't happen, though) It then returns false.
 - Otherwise, if the comment ended the line, it returns true.
- Otherwise, it assumes the token gotten was an argument and passes it to the ParseArg function. It sets parse_arg_succeeded equal to ParseArg(current_token, tokenizer, arg1, error_printer).
- If the function is still going, then it has parsed the last argument. It then returns the conjunction of parse_arg_succeeded and ParseOtherArgs (tokenizer, arg2, error_printer).

CSE 560N

76

ParseOtherArgs

- In this function, try to get the next token from tokenizer.
- If there isn't one, return true.
- If it is a COMMA, get another token.
 - If there isn't one, add the error code to the error_printer about expecting another argument after a comma.
 - If there is and it is a WHITESPACE, get another token and call it something like current_token. If there isn't one, add the error code to the error_printer about expecting another argument after a comma.
 - If it wasn't a WHITESPACE (or if it was and we got another token), now we are going to assume it is an argument. Set a boolean equal to ParseArg(current_token, tokenizer, arg, error_printer). Now we need to parse the other args. We know that in our program, we can only take up two arguments, and we already certainly have. So create a new Token object called something like 'dummy' to pass to the next call of ParseOtherArgs, since we don't want to overwrite the arg variable. Create another boolean and set it equal to ParseOtherArgs(tokenizer, dummy, error_printer). If the text field of dummy isn't empty after this call, add the error code to the error_printer about this version of the program only accepting instructions with two or less arguments.
- Finally, return the conjunction of the two boolean variables.
- If it is a WHITESPACE, get another token.
 - If there isn't one, return true.
 - If it's a COMMENT, try to get another token. There shouldn't be, but if there is one, add the error code to the error_printer about expecting nothing or a line after a comment. If there was another token, return false. If there wasn't, return true.
 - If it's anything else, add the error code to the error_printer about expecting a comment or nothing after a whitespace following an argument. Then return false.
- If it is anything else, add the error code to the error_printer about expecting a comma, whitespace, or nothing after an argument. Then return false.

CSE 560N

77

Literals

- Literals:
 - Instructions may reference literals rather than a symbolic label or absolute address. Literals can be integer numbers, hexadecimal, binary, hex or characters. They have the following format:
 - ADD 7,='ABCD' Character string max.
 - ADD 7,=153

Literals are not valid on transfers, shifts's, rotates read's, commands or directives

CSE 560N

78

Assembler Output

- Assembly Listing - Report to User
 - Complete copy of the users source line
 - Any errors to be reported appear immediately after the line in question
- Object file - Information for the LINKER
- Symbol Table - List of all variables defined or referenced in the program

CSE 560N

79

Big Assemblers are different

- Symbol Tables
 - add reference locations
 - length
 - data type
- MOT
 - Instruction lengths
 - Types - data types

CSE 560N

80

Big Assemblers are different

- POT
- Intermediate files

CSE 560N

81

Assemblers and passes

- One pass
 - forward referencing a problem
- Two pass
- Three pass
- N pass

CSE 560N Spring 2009

82

Table Sorting/Searching

- MOT (machine operations table)
- DOT (directives operations table)
 - Normally arranged based on most frequently used
- Symbol Table
 - sequential

CSE 560N Spring 2009

83

Star addressing & Expressions

- Star Addressing:
 - Use * as the current instruction address during assembly
 - `JMP 7, *+6` would BRanch to location LC+6
- Expressions:
 - The assembler supports the use of expressions as operands in the `equ` pseudo-op.
 - The arithmetic operations are limited to `+, -, *, /` with no more than two operations per expression.

CSE 560N

84

Literals

- 4 basic styles in order of implementation
 - In line
 - In line no repetition
 - Immediate
 - Pooled
 - at end
 - anywhere

CSE 560N

85

In line

Programmer Code Assembler Generated Code

```

ADD 7,=32           ADD 7,++2
Sub 7,MUD           * jmp ++2
                   * Num 32
                   sub 7,MUD
    
```

Second request for the same value

```

ADD 7,=32           ADD 7,++2
Sub 7,MUD           * jmp ++2
                   * Num 32
                   sub 7,MUD
    
```

CSE 560N

86

In line No repetition

Programmer Code Assembler Generated Code

```

ADD 7,=32           10 ADD 7,++2
Sub 7,MUD           *11 jmp ++2
                   12 Num 32
                   13 sub 7,MUD
    
```

Second request for the same value

```

ADD 7,=32           ADD 7,12
Sub 7,MUD           Sub 7,MUD
    
```

Literal Table
Value #32 location 12

CSE 560N

87

Immediate

uses this technique the literal is placed in the normal address field. We need a flag to indicate how the field has been used or a special opcode

Add 7,32 generates 00100000

But the size of the literal is limited—sure we could add a byte but still we have limited the size of a literal

Add 7,4000 generates 10100000

Digital and Intel solved this program by growing the word byte by byte until the entire literal can fit.

CSE 560N

88

Pooled at end

One idea is to have the literal follow the same technique that a programmer must do..ie define a word of memory using the dec pseudo-op. It is assigned LC/Memory location 60.

	LC		
	30	add	7,\$lit32
ADD	31	sub	7,MUD
	...		
Sub	...		
	60..	\$lit32	num 32

	Literal Table	
	\$lit32	addr=60

Pooled anywhere

- The programmer should have the right to define exactly where the literal pool should go.
- New directive "LORG" literal origin

Assembler History

Advantages of Assembly



Advantages of Assembly

- Over machine code (lower level)
 - easier to remember mnemonic operations than actual opcodes
 - e.g., ADD, SUB, MUL, DIV, ...
 - vs., 04, 2C, F6, F6, ...
 - similarly for addresses in program
 - e.g., BR LOOP1
 - vs., BR 465B

CSE 560N

93

Advantages of Assembly II

- Over higher-level languages
 - access to full capabilities of the machine
 - e.g., testing overflow flag, test-and-set instruction, ...
 - how would you do that in JAVA or PERL performance ... ? ...

CSE 560N

94

The “Best” of Both Worlds

- Systems programming is often done in a language like C/C++/C#
 - syntax of a higher-level (problem-oriented) language
 - but gives access to low-level machine, like assembly language

CSE 560N

95

An Old Myth

- “If a program will be used a lot, it should (for efficiency) be written in assembly.”
- No longer true!

CSE 560N

96

- Good compilers
- Fast machines
- Hard to write
 - 10 lines of code / day, independent of language
- Hard to read
 - high cost of maintenance
 - can be 2/3 of total
 - 15% programmer turnover

CSE 560N

97

Modern Approach

- Write in high-level language
- Analyze to find where time spent
- Invariably, it's a *small* part of the code
- Tune that tiny part for high performance
 - perhaps by writing in assembly language

CSE 560N

98

Modern Approach II

- Higher level can be a *performance* win too!
 - problem-oriented language gives problem-level insights
 - **huge** performance gains are in *algorithmic* insights
 - e.g., $O(n^3)$ vs $O(n \lg n)$
 - assembly language programmer tends to be immersed in bit-twiddling (saves small amounts all over, but misses big picture)

CSE 560N

99

Modern Approach III

- Conclusion:
 - assembly language use is often a holdover from when machines were *expensive*, and people were *cheap*

CSE 560N

100

So Why Do We Learn This Stuff?

- You may still need to write that tiny, critical part in assembly language
- Concepts/techniques similar for compilers
- Good vehicle for understanding architecture
- Legacy code with large parts in assembly language
- **This world still needs assemblers!**
 - many compilers translate to assembly language

CSE 560N

101

Operating Systems - Introduction

- When does a program become a process?
 - when it is assigned certain system resources
 - e.g., processor, memory, I/O, registers, ...
- At any instant, there are many processes
 - multiple, concurrent users
 - mix of batch and interactive jobs
 - OS tasks (e.g. spooling)
- But only a fixed number of resources...

CSE 560N

102

OS - Introduction (II)

- Resources must be *managed*
- This is the job of the *operating system* (OS)

CSE 560N

103

Challenges in OS

- Concurrency is fundamental
- Concurrency is hard
 - Example: sharing a bridge
 - Long tunnel that only fits 1 lane of traffic
 - What policy do you use to control traffic?
 - Example:
 - Process A is using X and needs Y,
" B " " Y " " X
 - OS must avoid *deadlock* and *starvation*

CSE 560N

104

Responsibilities of OS

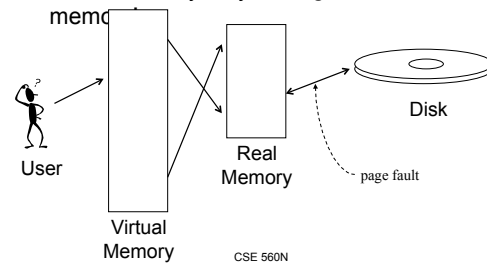
- Handles interrupts
 - may be generated by I/O or by programs
- Manages real memory
 - loading of segments
- Manages virtual memory
 - “virtual”: appears to user to have different characteristics than it has in actuality
 - virtual memory: a large block of contiguous memory space

CSE 560N

105

Responsibilities of OS (II)

- virtual memory may be *larger* than real



CSE 560N

106

Responsibilities of OS (III)

- File management
 - keeps file handles and position marks
- CPU
 - schedules processes (i.e. ready / waiting / idle)
- Security
 - prevents one user from damaging another's data
 - prevents user from damaging operating system

CSE 560N

107

Macro Processors

CSE 560N

108

Introduction

- “*Macro*”: a notational convenience for programmers
 - short-hand for commonly used blocks of code
 - not restricted to assembly languages
- “*Macro Processor*”: tool that replaces short-hand with corresponding block of code
 - performs string substitution (“expansion”)
 - no analysis of instructions
 - no semantics of programming language

CSE 560N

109

Example

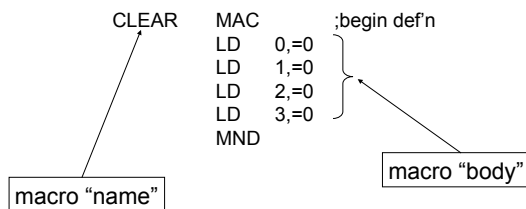
- To clear all registers, we write:


```
LD 0,=0
LD 1,=0
LD 2,=0
LD 3,=0
```
- If needed often, this can be tedious
- Solutions:
 - define a function and call it when needed
 - use a macro...

CSE 560N

110

Example II



CSE 560N

111

Example III

- In body of program:


```

...
CLEAR
...
CLEAR
...

```
- After being fed to macro processor:


```

...
LD 0,=0
LD 1,=0
LD 2,=0
LD 3,=0
...
LD 0,=0
LD 1,=0
LD 2,=0
LD 3,=0
...

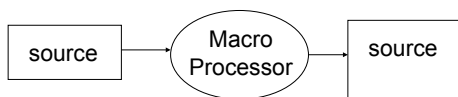
```

CSE 560N

112

Picture

- Notice that result is a larger source program



- Both programs are in the *same* language
 - i.e., same level of abstract machine

CSE 560N

113

Outline

- Features
 - arguments
 - labels
 - variables
 - conditional expansion
- Algorithm for macro processor
- Macros in C

CSE 560N

114

Macro Arguments

- Arguments make macros more flexible
- Involves *textual* substitution

```

SWAPMAC (&A,&B)
LD 1,&A
LD 2,&B
ST 1,&B
ST 2,&A
MND

Prog ORI
X dword 10
Y dword 0
SWAP (X,Y)
..
..
HALT
END
  
```

CSE 560N

115

Result of Macro Processing

```

Prog ORI
X dword 10
Y dword 0
LD 1,X
LD 2,Y
ST 1,Y
ST 2,X
..
..
HALT
END
  
```

CSE 560N

116

Labels and Macros

- Labels inside macro bodies can be useful
 - e.g., a macro that swaps values of 2 registers:

```

SWAPR      MAC      (&r1, &r2)
           BR      $Strt
$Tmp1     RES      1
$Tmp2     RES      1
$Strt     ST      &r2, $Tmp2
           LD      &r1, $Tmp2
           LD      &r2, $Tmp1
           MND

```

CSE 560N

117

Labels: Problem

- Consider a program with multiple invocations of macro SWAPR:

```

           ⋮
           SWAPR (1,2)
           ⋮
           SWAPR (1,3)
           ⋮

```

- Expands to:

- Labels defined twice!
 - assembler error

CSE 560N

118

Labels: Solution

- Macro processor provides a mechanism for generating unique labels
 - e.g., preface symbol (definition and use) with \$

```

SWAPR      MAC      (&r1, &r2)
           BR      $Strt
$Tmp1     RES      1
$Tmp2     RES      1
$Strt     ST      &r2, $Tmp2
           LD      &r1, $Tmp2
           LD      &r2, $Tmp1
           MND

```

CSE 560N

119

Labels: Solution II

- First expansion of this macro:


```

           BR      $AAStrt
$AATmp1   RES      1
$AATmp2   RES      1
$AAStrt   ...

```
- Unique prefix for each invocation
 - generated symbols must conform to assembler syntax (begins with \$, length, etc.)
 - programmer follows conventions (not to use \$ outside of macros, use short labels, etc.)

CSE 560N

120

Variables

- Evaluated at time of: expansion
 - i.e., *not* at execution time
- Example: `&Test SET 0`
 - variable name
 - special pseudo-op
 - expression
- `&Test` can then be used in expressions within the macro body
- This feature is often used in conjunction with...

CSE 560N

121

Conditional Expansion

- So far, all macros we've seen have been expanded to the *same* block of code
 - (modulo argument replacement)
- Useful to generate *different* blocks of code
 - perhaps depending on value of some bool expr
- Syntax: IF / ELSE / ENDIF

```
IF (expr)
block1
ELSE
block2
ENDIF
```

Meaning:
if expr is true, expand with block 1
else, expand with block 2

CSE 560N

122

Example: Shifting Left/Right

```
SHIFT MAC (&Target, &Amount, &Dir)
BR $Srrt
$Tmp RES 1
$Srrt ST 1, $Tmp
LD 1, &Target
IF (&Dir EQ 0)
SHL 1, &Amount
ELSE
SHR 1, &Amount
ENDIF
ST 1, &Target
LD 1, $Tmp
MND
```

123

124

Example: Swap

- Conditional expansion for efficiency:


```

            SWAPMAC (&A,&B)
                IF (&A NEQ &B)
                    LD 1,&A
                    LD 2,&B
                    ST 1,&B
                    ST 2,&A
                ENDIF
            MND
            
```
- Now SWAP (S,S) is "expanded" to nothing

125

Macros vs. Functions: Tradeoffs

- | | |
|--|---|
| <ul style="list-style-type: none"> • Macros are expanded inline • Disadvantage: <ul style="list-style-type: none"> – program size increases • Advantage: <ul style="list-style-type: none"> – speed | <ul style="list-style-type: none"> • Functions are called (branched to) • Disadvantage: <ul style="list-style-type: none"> – overhead of a branch instruction (more expensive than func!) • Advantage: <ul style="list-style-type: none"> – program size |
|--|---|
- This is another example of the space / time tradeoff

126

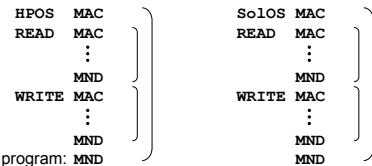
Algorithm: First Attempt

- 2-pass approach is tempting
 - lets us resolve forward references
- 1st pass:
 - build table of key, domain: Macro names and attribute, range: Macro bodies
- 2nd pass:
 - do the expansion (replace macro calls with bodies)
- 1st pass Invariant: after each MND, table contains all previous macro names seen in definitions, and their bodies

127

Problem A: Nested Definitions

- Often useful to define macros *inside* macros



- In program: MND
 - begin with OS macro (e.g. HPOS)
 - then use READ & WRITE

128

Nested Definitions

- To recompile on different OS, change flag at the top of program only!
- Another solution?
 - Conditional expansion
 - but nested definitions more convenient. Why?
- Will this work with our 2-pass approach?
 - multiple definitions of "READ"
 - (notice how invariant is violated)
- Problem: "definitions" depend on previous "expansions"

129

Algorithm: Second Attempt

- Use a 1-pass approach that alternates, as necessary, between defining and expanding
- Data structures:
 - DefTable -- macro definitions
 - NameTable -- macro names
- Key invariant: after each "outer" MND seen
 1. All previous "outer" macro definitions have been inserted into the table
 2. All previous macro invocations expanded

CSE 560N

130

Algorithm: Intuition

- Scan program line-by-line
- MAC seen: change into "definition mode"
 - insert body into DefTable
 - match up *outer* MND with *initial* MAC
- Macro call seen: change into "expansion mode"
 - look up macro name in table
 - process expansion from DefTable line-by-line
 - may include macro definitions!!
 - requires changing back into "definition mode"

CSE 560N

131

Algorithm: Limitation

- This two-edged approach is pretty clever...
- But are there any limitations it imposes on the definition / use of macros?
- A. Yes:
 - I.e: No forward references
- In practice, this is not a big problem

CSE 560N

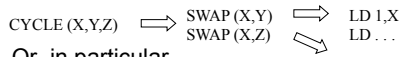
132

Problem B: Nested Invocations

- Convenient to allow macros to call macros

```
CYCLE      MAC (&A,&B,&C)
           SWAP (&A,&B)
           SWAP (&A,&C)
           MND
```

- Expansion requires further expansion



- Or, in particular . . .

CSE 560N

133

Recursive Invocations

- Macro invokes itself!
- Of course, beware infinite recursion:

```
TROUBLE MAC
        NMD 10
        TROUBLE
        MND
```

- Solution: Use conditional expansion

CSE 560N

134

Recursive Macro: Example

- Consider

```
TAB MAC (&C)
    IF (NZ &C)
    TAB (&C-1)
    ENDIF
    NMD &C
    MND
```

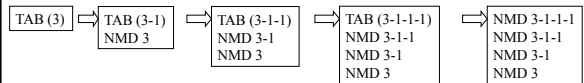
- Exercise: expand TAB (3)
- Exercise: what happens with

```
Depth EQU 3
TAB (Depth)
```

CSE 560N

135

Expansion of TAB (3)



CSE 560N

136

Algorithm: Refinement for Nested Invocations

- Add a data structure
 - ArgStack -- arguments in current expansion
- As nested expansions encountered:
 - arguments are pushed onto this stack
- Now, “expand mode” means:
 - look up macro name in table
 - *push arguments onto stack*
 - process expansion from DefTable line-by-line
 - may include macro definitions (change back to definition mode)
 - *may include macro invocations (expansions)*
 - After last line from DefTable, *pop arguments off stack*.

CSE 560N

137

Macros in C

CSE 560N

138

MP Algorithmic Highlights

- Multiple passes
 - Forward references ok!
- BUT, no recursion allowed
 - Self-references not further expanded
 - `#define T (x+T)` //only one expansion of T
 - Circularities handled the same way (stop at first self-reference)
- First action: strip comments
- View results of macro expansion with `-E`
 - `gcc -E test.c > test.i`
 - Standard file extension for preprocessed C (C++) is `.i` (ii)

CSE 560N

139

Basic Features: Definition

- Use `#define`
 - e.g., `#define BUFF_SIZE 1000`

macro name macro body
- Must be 1 line
 - longer definitions use “line continuation”, `\`
- Naming convention: all upper case
- Defines a global constant
 - e.g. of use: `int Buffer[BUFF_SIZE];`
 - to change this constant, must recompile

CSE 560N

140

Using Arguments

- Argument list follows name (no space):
 - `#define INC(X) X++`
 - `#define SUM(X,Y) X+Y`
- **DANGER:** “arithmetic grouping”
- Problem #1: protecting the body
 - e.g., `#define MAX(X,Y) X > Y ? X : Y`
 - works fine with `a = MAX(b,c);`
 - but consider `a = MAX(b,c) + 1;`
- solution: protect the *body*

```
#define MAX(X,Y) (X > Y ? X : Y)
```

CSE 560N

141

Using Arguments II

- Problem #2: protecting the arguments
 - now consider using MAX macro in:


```
flag = MAX (b>0, c<0);
```
 - i.e.
 - solution: protect the *arguments*

```
#define MAX(X,Y) ((X) > (Y) ? X : Y)
```
- Aside: line continuation
 - `#define INC(X,Y) \`
`{(X)++; \`
`(Y)++; }`

CSE 560N

142

Conditionals

- Common condition “is this macro defined?”


```
#ifndef BUFF_SIZE
#define BUFF_SIZE 1000
#endif /*BUFF_SIZE*/
```
- Application: debugging modes


```
#define DEBUG_ON

#ifdef DEBUG_ON
printf ( . . . );
#endif /*DEBUG_ON*/
```

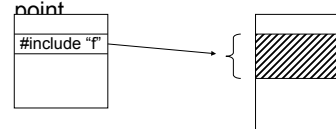
Incurs **no**
space/time
overhead when
not debugging!

CSE 560N

143

File Inclusion

- Syntax: `#include “filename”`
 - text of file called *filename* inserted at that point



- **DANGER:** recursive inclusion

File f1
#include “f2”

File f2
#include “f1”

144

Recursive File Inclusion

- Solution: protect every included file
- Convention: use `#ifndef ... #endif`

File *F1.h*

```

#ifndef F1_H_PAGS
#define F1_H_PAGS
...
#endif /*F1_H_PAGS*/

```

filename → `F1_H_PAGS`
something unique → `/*F1_H_PAGS*/`

CSE 560N

145

Predefined Macros

- Some defined by ANSI standard:
 - `__FILE__` / `__LINE__`: current file name / line number
 - `__DATE__` / `__TIME__`: current date / time
- Useful for error reporting


```
printf("error in %s, line %d\n",
      __FILE__, __LINE__);
```
- Others defined by particular compilers (e.g., gcc)
 - `__VERSION__`, `__BASE_FILE__`, `__INCLUDE_LEVEL__`
- Useful for distinguishing OS's


```
#ifdef __VAX__
...
#endif /*__VAX__*/
```

CSE 560N

146

Arguments in Strings

- ANSI C: parameter substitution not performed within quoted strings
 - `#define DISP(EXP) printf("EXP = %d\n", EXP)`
 - Invocation: `DISP (i*j+1);`
 - Result: `printf ("EXP = %d\n", i*j+1);`
- Solution: "stringizing" operator, `#`
 - `#define DISP(EXP) printf(#EXP "= %d\n", EXP)`
 - Result: `printf ("i*j+1" "= %d\n", i*j+1);`

CSE 560N

147

Pitfalls to Avoid

- "Text substitution" aspect of macros can make them tricky
- General strategy: limited use!
- Pitfall #1: side effects
 - recall MAX example
 - consider: `a = MAX(b++, c++)`
 - Q. if `b = 2`, `c = 5` beforehand, what is result?
 - A. `a = 6` `b = 3` `c = 7`

CSE 560N

148

Pitfalls to Avoid II

- Pitfall #2: swallowing the semicolon
 - Macro expands to form a compound statement:
`#define INC(X,Y) {X++; Y++;}`
 - It is tempting to include semicolon with call:
`INC(a,b);`
 - But consider:
`if (...) INC(a,b); else ...`
 - This doesn't compile! why not?
 - Solution (notice the "missing" semicolon at the end):
`#define INC(X,Y) \`
`do { \`
 `X++; Y++;; \`
`while(0)`

CSE 560N

149

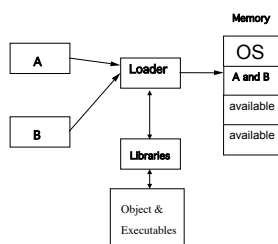
Linkers and Loaders

CSE 560N

150

Loader Basic Functions

- Relocation
- Allocation
- Linking
- Load Code



CSE 560N

151

Linking

- Since programs are not always compiled or assembled at the same time, we have to deal with references to variables defined in other programs. The program in which the variable is defined must declare the variable as being shared (entry) with other programs. The program that references the variable must tell the assembler or compiler that the variable is in an external program (extrn).
- The linker/loader must use the information to build a symbol table of all the shared variables and their locations so that EXTERNAL references can be resolved (linked).

CSE 560N

152

Relocation

- When a program is assembled/compiled there is no way that the translator will know the memory location that will be assigned the executable. All addresses are therefore relative to an assumed starting point. All memory assignments and references must be adjusted when the program is ultimately placed in memory. These address relocations (adjustments) are critical.
- An instruction may have 2 addresses to relocate
 - 1. The physical address assigned to the instruction. Typically resolved by adding the difference between the operating system assigned address and the translator assigned address
 - 2. If the instruction references a location within the program, then this reference will need the same adjustment.

CSE 560N

153

Allocation

- The linker/loader must compute the overall memory requirements of the program. The information is either then handed directly to the operating system, or provided in a header record in an executable file.

CSE 560N

154

Translators and Linkers/ loaders

- They must work together
 - Deliver straight forward information
 - Linker should not require extensive parsing and discovery
 - Format of the data is key

CSE 560N

155

Loaders Why do we need them

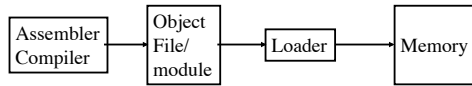
- Lazy Programmers
- Management request
- Productivity

Now some history.....

CSE 560N

156

Loaders-Absolute



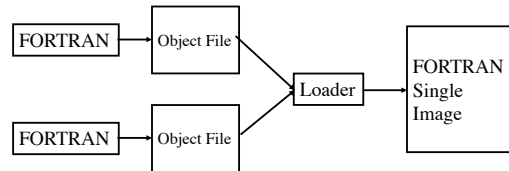
Loaded exactly where the assembler assumed (or programmer specified) the program would be loaded.

- Problems
 - Memory management
 - compilation issues

CSE 560N

157

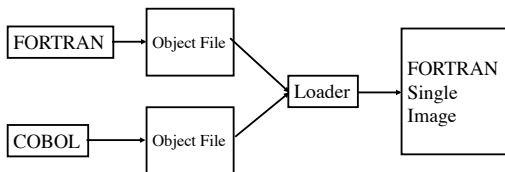
Loaders



CSE 560N

158

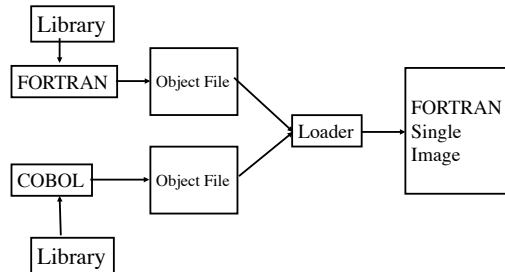
Loaders Multiple Languages



CSE 560N

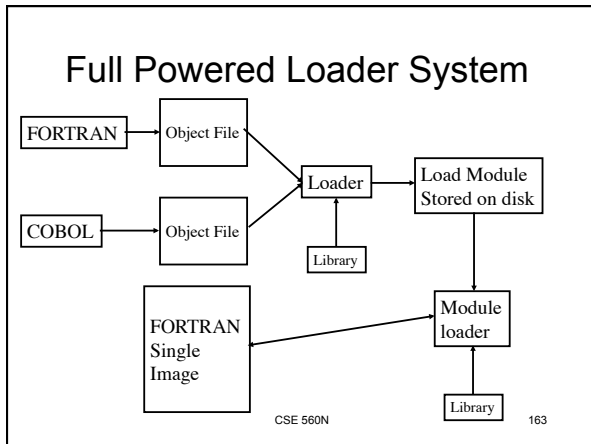
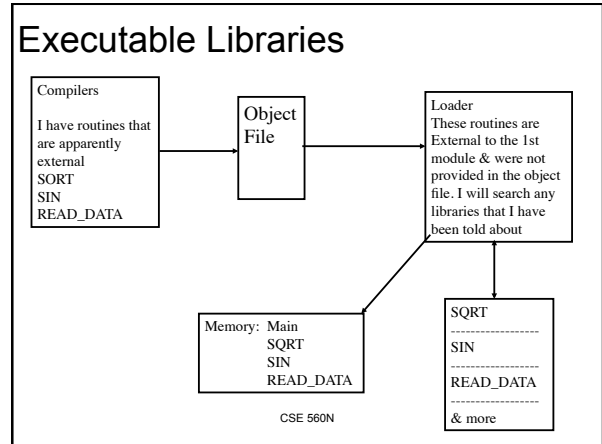
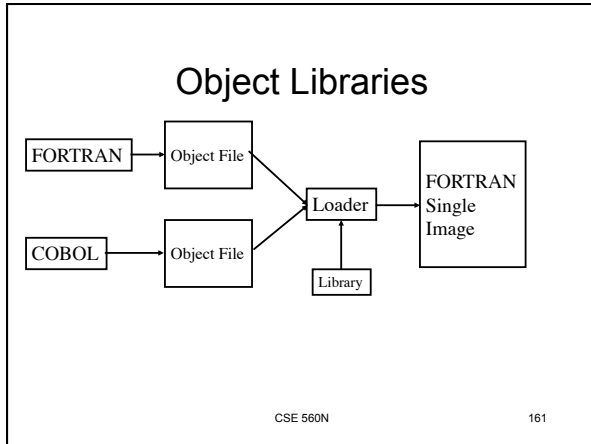
159

Source Libraries

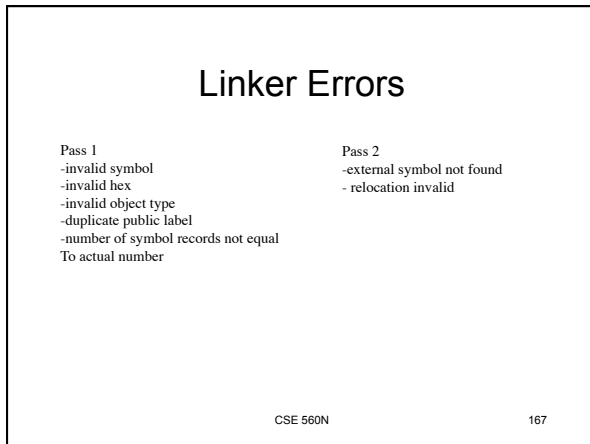
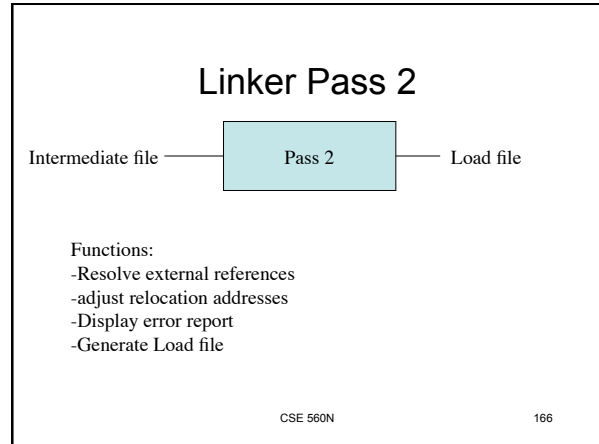
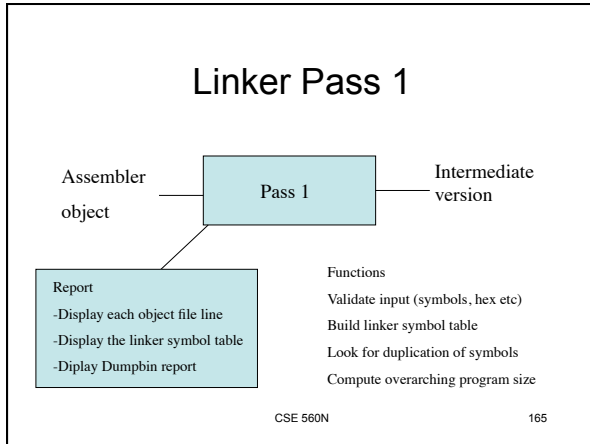


CSE 560N

160



- ### Object files and Formats
- A variety of techniques have been tried.
 - Vendor specific IBM, HP, Msoft
 - Generic UNIX elf
 - Subset of executable
 - Fixed format
 - Variable format with built in descriptors
- CSE 560N 164



```

1  0  PGA  START
2      ENTRY  A1,A2
3      EXTERNAL B1,PGB
4  20  A1
5  30  A2
6  40          ADDR  A(A1)
7  44          ADDR  A(A2+15)
8  48          ADDR  A(A2-A1-3)
9  52          ADDR  A(PGB)
10 56          ADDR  A(B1+PGB-A1+4)
    
```

Variable	Type	addr	Length	Reference
PGA	Pgm Name	0	60	1
A1	Local var	20	n/a	2
A2	Local var	30	n/a	2
PGB	Ext Var	n/a	n/a	3
B1	Ext Var	n/a	n/a	3

Relative Address	Contents	What the assembler did	Reference
40	20	=20	6
44	45	=30+15	7
48	7	30-20-3	8
52	0	unknown	9
56	-16	-20+4	10

CSE 560N 168

```

1  0  PGA  START
2  ENTRY  A1,A2
3  EXTERNAL B1,PGB
4  20  A1  EQU *
5  30  A2  EQU *
6  40  ADDR A(A1)
7  44  ADDR A(A2+15)
8  48  ADDR A(A2-A1-3)
9  52  ADDR A(PGB)
10 56  ADDR A(B1+PGB-A1+4)

```

• RLD Records

Relative Address	Operator	Variable Name	Reference to source line
40	+	PGA	6
44	+	PGA	7
52	+	PGB	9
56	+	B1	10
56	+	PGB	10
56	-	PGA	10

CSE 560N Spring 2009 169

```

• 12 0  PGB  START
13  ENTRY  B1
14  EXTERNAL A1,A2
15  16  B1
16  24  ADDR  A(A1)
17  28  ADDR  A(A2+15)
18  32  ADDR  A(A2-A1-3)

```

Variable name	Type	Address	Length	Reference
PGB	Pgm name	0	38	12
B1	Local Variable	16	n/a	13
A1	External Var	Unknown	n/a	14
A2	External Var	Unknown	n/a	14

Relative address	Contents	What the assembler did	Reference
24	0	=0 A1 unknown	16
28	15	=15 A2 unknown	17
32	-3	=-3 A1&A2 unknown	18

Relative address	Operator	Variable	Reference
24	+	A1	16
28	+	A2	17
32	+	A2	18
32	-	A1	18

170

Final memory address	Contents at the memory location	Notes	Reference
104		Load pt PGA	
144	124		6
148	149		7
152	7		8
156	168		9
160	232		10
164	unused		
168		Load pt PGB	
192	124		16
196	149		17
200	7		18

CSE 560N 171

Writing a Hardware Simulator

CSE 560N 172

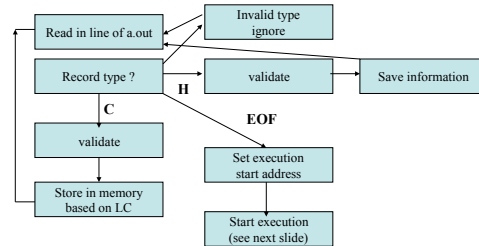
Writing a simulator

- Design
 - Process Input
 - Input validation
 - Load into an array/structure set up to be the size of memory 256 words (0:255)
 - Determine Execution start address
 - Simulation
 - Process each word of memory that the LC is pointing at
 - Capture and report all execution errors
 - Overflow
 - Division by zero
 - Shift 16 bits
 - LC > than 1023
 - LC out side the range of the program
 - Tight infinite loop

CSE 560N

173

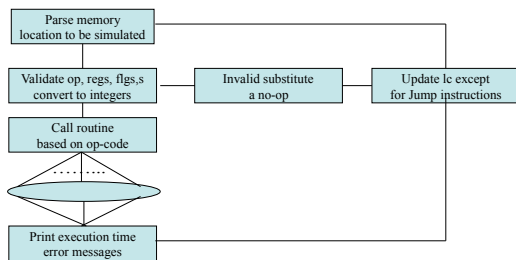
Simulation Design 1



CSE 560N

174

Simulation Design 2



CSE 560N

175

Simulation Design 3

```

    Procedure ADD
    DECLARE TEMP Double long
    TEMP= top of data stack+MEM(EFFADD)
    call test_overflow(TEMP)
    if OK=1 then top of data stack=TEMP
    return
    end

    Procedure test_overflow(OTEMP)
    DECLARE OTEMP Double long
    if OTEMP < -2^15 or > 2^15-1
    then print Overflow
    else OK=1
    return
    end
    
```

CSE 560N

176

Simulation Design 4

```

Procedure DIV
DECLARE TEMP Double long
if mem(effadd) = 0
then set statusword flag for overflow

else TEMP= top of data stack / mem(effadd)
call test_overflow(TEMP)
if OK=1 then top of data stack =TEMP
return
end

```

CSE 560N

177

Overflow

- Add
- Subtract
- Multiply
- Divide
- Read Integer

CSE 560N

178

Other Errors

- Division by zero
- Shift greater than 32 bits
- LC outside of range
- EFFADD outside of range
- Tight infinite loop

CSE 560N

179

EFFADD & JMP

```

FUNCTION Procedure EFFADD(Sfield,xfield)
Declare ETEMP Integer
If addr-flgs=0 then
If xfield=0 then ETEMP=Sfield
else ETEMP=Sfield+REG(Xfield)
if ETEMP not within range of pgm
then set status word EFFADD out of range
else EFFADD=ETEMP
Else.....:needs cases for each addr-flg type
return
end
FUNCTION Procedure JMP
LC=EFFADD
return
end

```

CSE 560N

180