成功大學
National Cheng Kung University

Compiler Course Program Assignment

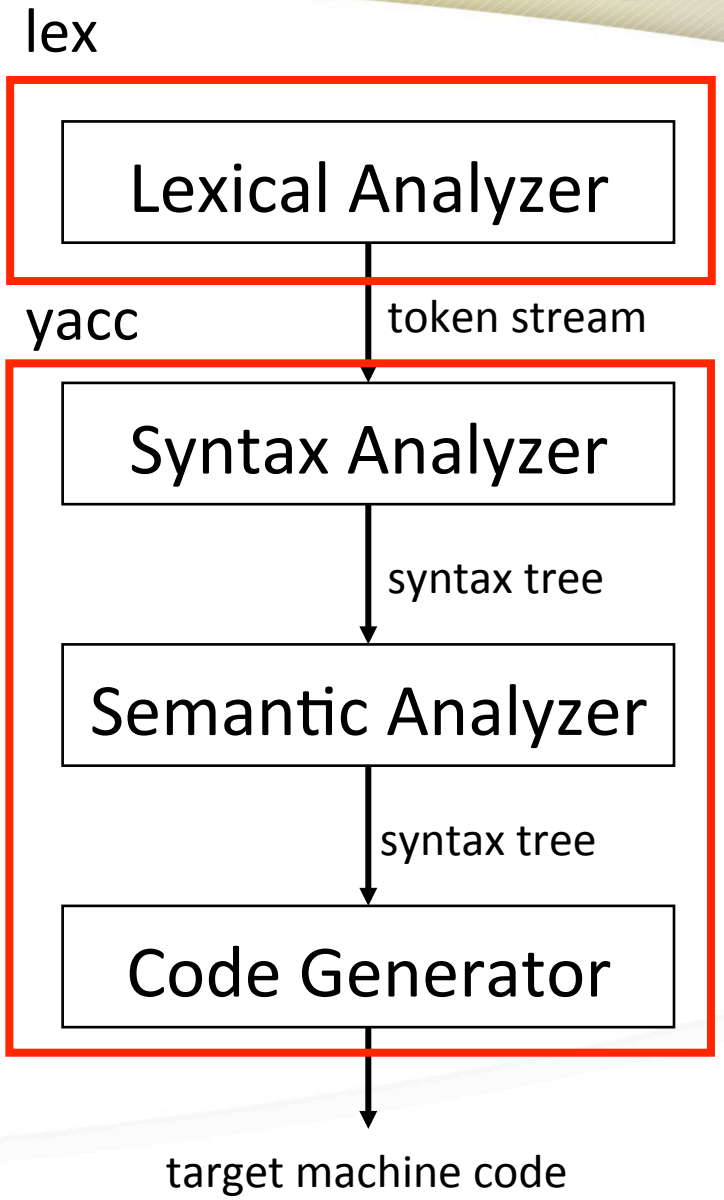
Speaker: Jeff Liaw

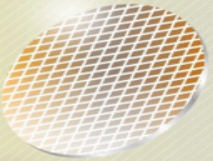
Program Assignment

- Goal: Build a compiler for C
- Deadline: 6/16 23:40
- Demo: 6/17
- Upload your project to moodle
  - A zipped file(.rar, .zip, .7z, ...) contain your source code and readme
    - do not hand in corpse(e.g. can not be compiled or be ran)!
  - Filename: StudentID
    - e.g. F74012345.rar
- Post your problems on moodle or ask TAs
- Environment: Ubuntu 14.04 with gcc 4.8.4

Compile Process

- Aid tools:
  - lex(flex 2.5.35)
  - yacc(bison 3.0.2)
- Lex is a program generator designed for lexical processing of character input streams
- Yacc provides a general tool for imposing structure on the input to a computer program





Token Format(1)

- Keywords
  - `int char return if else while break print read`
- Arithmetic Operators
  - `= ! + - * /`
- Comparison Operators
  - `== != < > <= >= && ||`
- Special Symbols
  - `[ ] ( ) { } ; ,`

Token Format(2)

- Identifier
  - 所有Identifier都以id為開頭，後面接一串大寫開頭的英文字母
  - 字母字串開頭為大寫，其他小寫，長度不限
  - 正確範例：idVariable, idApple, idFunction, idL
  - 錯誤範例：IdVariable, idapple, varid
- Number
  - 一串以0到9的digit組成的字串，長度不限
  - 正確範例：0, 3456, 655453
  - 錯誤範例：abc123

Token Format(3)

- Char
  - 雙引號中間夾任意一個字元
  - 正確範例：”a”, “3”
  - 錯誤範例：’a’, ‘321’, “123”
- Comment
  - 單行中，//以及//後面的內容都是註解
  - e.g. int main { // main function
- 其他都算是錯誤

Grammar

- 參考附件grammar.txt

head(left side)

body(right side)

production 1

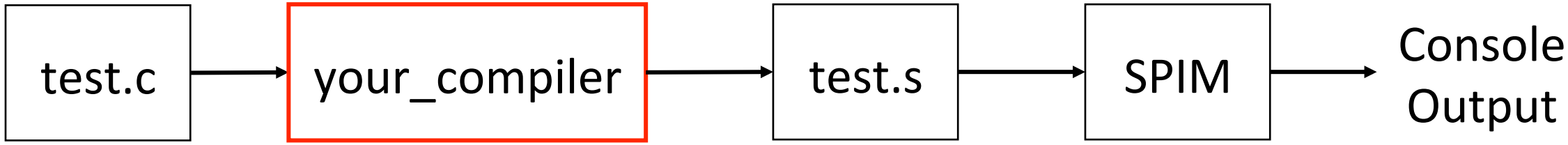
production 2

```
Program
    VarDeclList FunDeclList
VarDeclList
    VarDecl VarDeclList
    epsilon
VarDecl
    Type id ;
    Type id [ num ] ;
FunDeclList
    FunDecl
    FunDecl FunDeclList
FunDecl
    Type id ( ParamDeclList ) Block
```

Program → VarDeclList FunDeclList
VarDeclList → VarDecl VarDeclList | epsilon
VarDecl → Type id ; | Type id [num] ;
FunDeclList → FunDecl | FunDecl FunDeclList
FunDecl → Type id (ParamDeclList) Block

Expected Result

- Input: test.c
- Output: test.s
- Run test.s with SPIM, check the console output

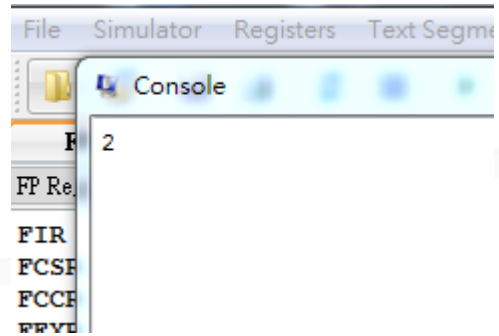


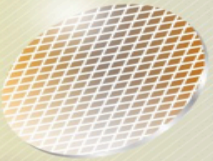

```
int main() {
    int x;
    x = 2;
    print x;
}
```

your_compiler

```
    .text
    .globl main
main:
    li $a0, 2
    li $v0, 1
    syscall
```

SPIM



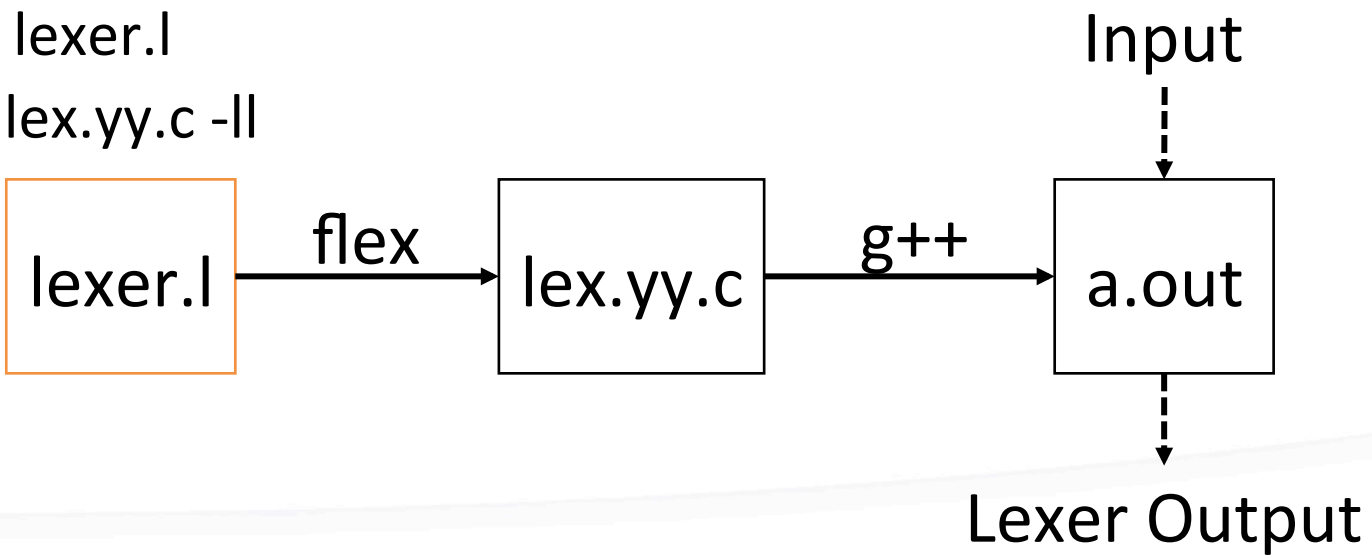


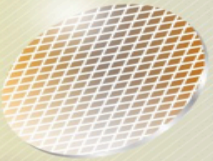
Grading

- IO: read/print - 10%
- Arithmetic - 10%
  - priority - 5%
- Conditional - 15%
- Loop - 15%
- Array - 15%
- Function call - 20%
- Demo - 10%

Introduction to Flex

- Flex allows you to implement a lexical analyzer by writing rules
- Flex compiles your rule file(e.g. lexer.l) to C source code implementing a finite automaton
  - you don't need to understand the C source file
- Usage
  - $ flex lexer.l
  - $ gcc lex.yy.c -ll





Rule Files

- Declarations and User subroutines sections are optional
  - write declarations and helper function in C
- Definitions section is also optional
  - useful to name regular expression
  - e.g. `DIGIT [0-9]`

Rule File Structure

```
%{
Declarations
%}

Definitions
%%
Rules
%%
User subroutines
```

Lex Regular Expression

x	the character “x”
“x”	an “x”, even if x is an operator
\x	an “x”, even if x is an operator
[xy]	the character x or y
[x-z]	the characters x, y or z
[^x]	any character but x
.	any character but newline
^x	an x at the beginning of a line
<y>x	an x When Lex is in start condition y

x$	an x at the end of a line
x?	an optional x
x*	0,1,2, ... instances of x
x+	1,2,3, ... instances of x
x|y	an x or an y
(x)	an x
x/y	an x but only if followed by y
{xx}	the translation of xx from the definitions section
x{m,n}	m through n occurrences of x

Operators
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

Lex Actions

- Rule specifies an action to perform if input matches the regular expression or definition
- The action is specified by writing regular C code
- The default action is copying input to output
- e.g.

```
[0-9] {
    printf("%s", yytext);
}
[ \t\n] ;
. ;
```

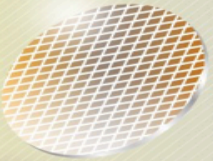
Rule File Structure

```
%{
Declarations
%}

Definitions
%%
Rules
%%
User subroutines
```

Lex Predefined Variables

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylineno`	current line number
`yylval`	value associated with token
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string



Calculator Example

- A lexical analyzer of simple calculator
  - only plus and multiply
  - only integer
- Token
  - Integer Constant: [0-9]+
  - Plus Operator: “+”
  - Multiply Operator: “*”

Declarations

- Between "%{" and "%}"
- Content between those is copied verbatim into output file

- A helper function to show the error message
  - we need to declare function prototype, so that we can use later

Code Section	

1
2
3
	%{
	void yyerror();
	%}

Rule File Structure

```
%{
Declarations
%}

Definitions
%%
Rules
%%
User subroutines
```

Definitions

- Name two useful regular expression
  - digit: [0-9]
  - int_const: {digit}+

Code Section	
1	digit [0-9]
2	int_const {digit}+

Rule File Structure

```
%{
Declarations
%}

Definitions
%%
Rules
%%
User subroutines
```

Rules

- Print some message during lexical analyzing
  - integer_const, plus and multiply

Code Section

```
1  {int_const} { printf(“ INTEGER_CONST %s”, yytext); }
2  “+”         { printf(“ PLUS ”); }
3  “*”         { printf(“ MULT ”); }
4
5  [ \t]*      {}
6  [\n]        { yylineno++; }
7
8  .           { fprintf(stderr, “LEXER ”); yyerror(); exit(1)}
```

Rule File Structure

```
%{
Declarations
%}

Definitions
%%
Rules
%%
User subroutines
```

User Subroutines

- Main program and error printer

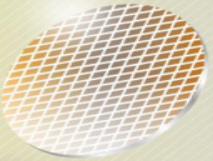
Code Section

```
1   int main() {
2       yylex();
3       return 0;
4   }
5
6   void yyerror() {
7       extern int yylineno;
8       extern char *yytext;
9
10      fprintf(stderr, “Error: at symbol \”%s\” on line %d\n”
11              , yytext, yylineno);
12  }
```

Rule File Structure

```
%{
Declarations
%}

Definitions
%%
Rules
%%
User subroutines
```

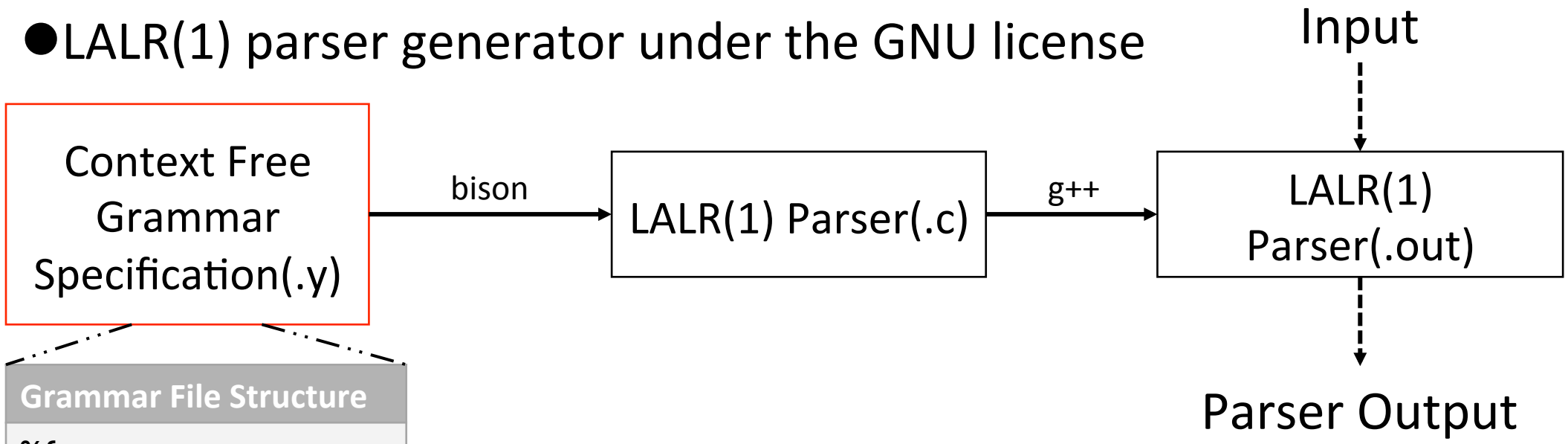


Reference

- http://flex.sourceforge.net/#overview
- http://dinosaur.compilertools.net/lex/index.html

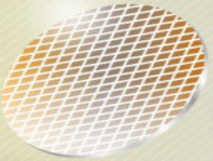
Introduction to Bison

- LALR(1) parser generator under the GNU license



Grammar File Structure

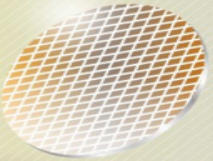
```
%{
Prologue
%}
Bison Declarations
%%
Grammar Rules
%%
Epilogue
```



Calculator Example

- We just implemented a simple number lexer in flex
- Now we can implement a parser which will take actions on this file
  - do real calculate!

- Usage:
  - $ flex calc.lex
  - $ gcc -c lex.yy.c
  - $ bison -d -v calc.y
  - $ gcc -c calc.tab.c
  - $ gcc lex.yy.o calc.tab.o -o calc



Prologue

- Between “%{“ and “%}”
- Content between those is copied verbatim into output file
- Declare functions prototype, so we can use later

Code Section	
1	%{
2	int yyerror();
3	int yylex();
4	%}

Grammar File Structure

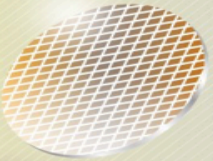
```
%{
Prologue
%}
Bison Declarations
%%
Grammar Rules
%%
Epilogue
```

Bison Declarations(1)

- Define the symbols used in formulating the grammar and the data of semantic values
- %token name - declare a token name (terminal symbol)
- %type symbol - declare value type of nonterminal symbol
- Specify precedence
  - lower the rule, the higher the precedence
  - %left symbol - left associativity
  - %right symbol - right associativity
- %union - specify the entire collection of possible data type for semantic values
- %start symbol - specify the start symbol

Grammar File Structure

```
%{
Prologue
%}
Bison Declarations
%%
Grammar Rules
%%
Epilogue
```



Bison Declarations(2)

Code Section

```
1   %union{
2       int     int_val;
3       string* op_val;
4   }
5
6   %start input
7
8   %token <int_val> INTEGER_LITERAL
9   %type  <int_val> exp
10  %left  PLUS
11  %left  MULT
```

Grammar File Structure

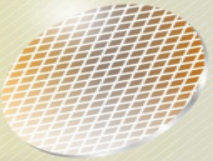
```
%{
Prologue
%}
Bison Declarations
%%
Grammar Rules
%%
Epilogue
```

Grammar Rules(1)

- A Bison grammar rule has the following generate form:
  - `result: components...;`
  - `e.g. exp: exp ‘+’ exp;`
- Can be followed by braces to indicates actions to take
  - `e.g. exp: exp ‘+’ exp { printf(“sum two numbers\n”); }`
- Multiple rules for the same result can be separately by vertical-bar character ‘|’
  - `e.g. result: rule1-components...`
    `| rule2-components...`
    `;`

Grammar File Structure

```
%{
Prologue
%}
Bison Declarations
%%
Grammar Rules
%%
Epilogue
```



Grammar Rules(2)

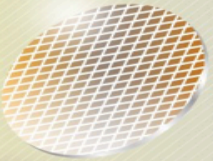
- Semantic values
  - value of component n is $n
  - modifiable lvalue is $$

Code Section

```
1  input: /* empty */
2         | exp { cout << “Result: ” << $1 << endl; }
3
4  exp:   INTEGER_LITERAL { $$ = $1; }
5         | exp PLUS exp { $$ = $1 + $3; }
6         | exp MULT exp { $$ = $1 * $3; }
```

Grammar File Structure

```
%{
Prologue
%}
Bison Declarations
%%
Grammar Rules
%%
Epilogue
```



Epilogue

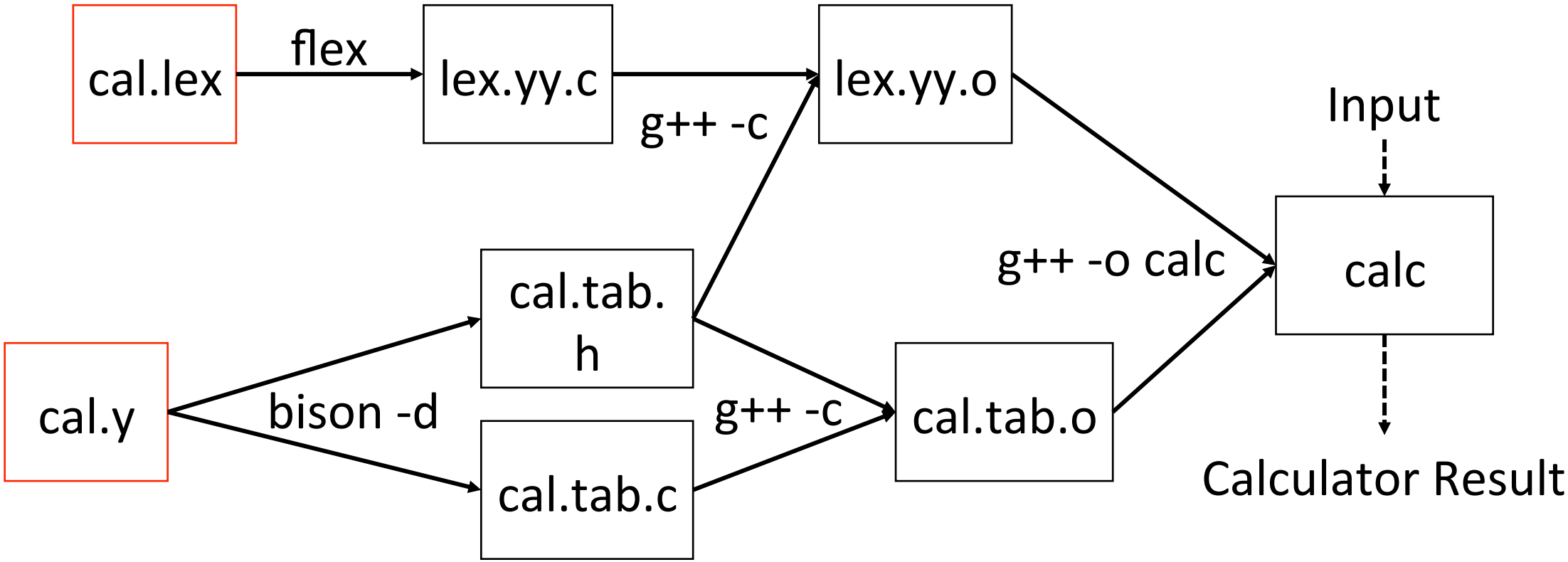
- Copy verbatim to the end of the output file
- Put your subroutine here

Code Section	
1	`int main() {`
2	`    yyparse();`
3	
4	`    return 0;`
5	`}`

Grammar File Structure

```
%{
Prologue
%}
Bison Declarations
%%
Grammar Rules
%%
Epilogue
```

Combine Flex and Bison



Pass Token from Flex to Bison

- Recall lex predefined variables
  - yylval

Filename: cal.tab.h

```
55 #if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
56 typedef union YYSTYPE
57 #line 7 "calc.y"
58 {
59     int     int_val;
60     string* op_val;
61 }
62 /* Line 1529 of yacc.c.  */
63 #line 64 "calc.tab.h"
64     YYSTYPE;
65 # define yystype YYSTYPE /* obsolescent; will be withdrawn */
66 # define YYSTYPE_IS_DECLARED 1
67 # define YYSTYPE_IS_TRIVIAL 1
68 #endif
69
70 extern YYSTYPE yylval;
```

Name	Function
int yylex(void)	call to invoke lexer, returns token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
char *yytext	pointer to matched string
yyleng	length of matched string
yylineno	current line number
yylval	value associated with token
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start condition
ECHO	write matched string

Pass Token from Flex to Bison

- lexer have to modify yylval and return token id
- parser will take the yylval and token id to make proper action - $$, $n

Filename: cal.tab.h

```
36 /* Tokens.  */
37 #ifndef YYTOKENTYPE
38 # define YYTOKENTYPE
39    /* Put the tokens into the symbol
40       know about them.  */
41    enum yytokentype {
42      INTEGER_LITERAL = 258,
43      PLUS = 259,
44      MULT = 260
45    };
46 #endif
47 /* Tokens.  */
48 #define INTEGER_LITERAL 258
49 #define PLUS 259
50 #define MULT 260
```

Code Section: calc.y(grammar rules segment)

```
1  exp:    INTEGER_LITERAL { $$ = $1; }
2          | exp PLUS exp { $$ = $1 + $3; }
3          | exp MULT exp { $$ = $1 * $3; }
```

Code Section: calc.lex(rules segment)

```
1  {int_const} { yylval.int_val = atoi(yytext); return INTEGER_LITERAL}
2  "+"         { yylval.op_val = new std::string(yytext); return PLUS }
3  "*"         { yylval.op_val = new std::string(yytext); return MULT }
```

Reference

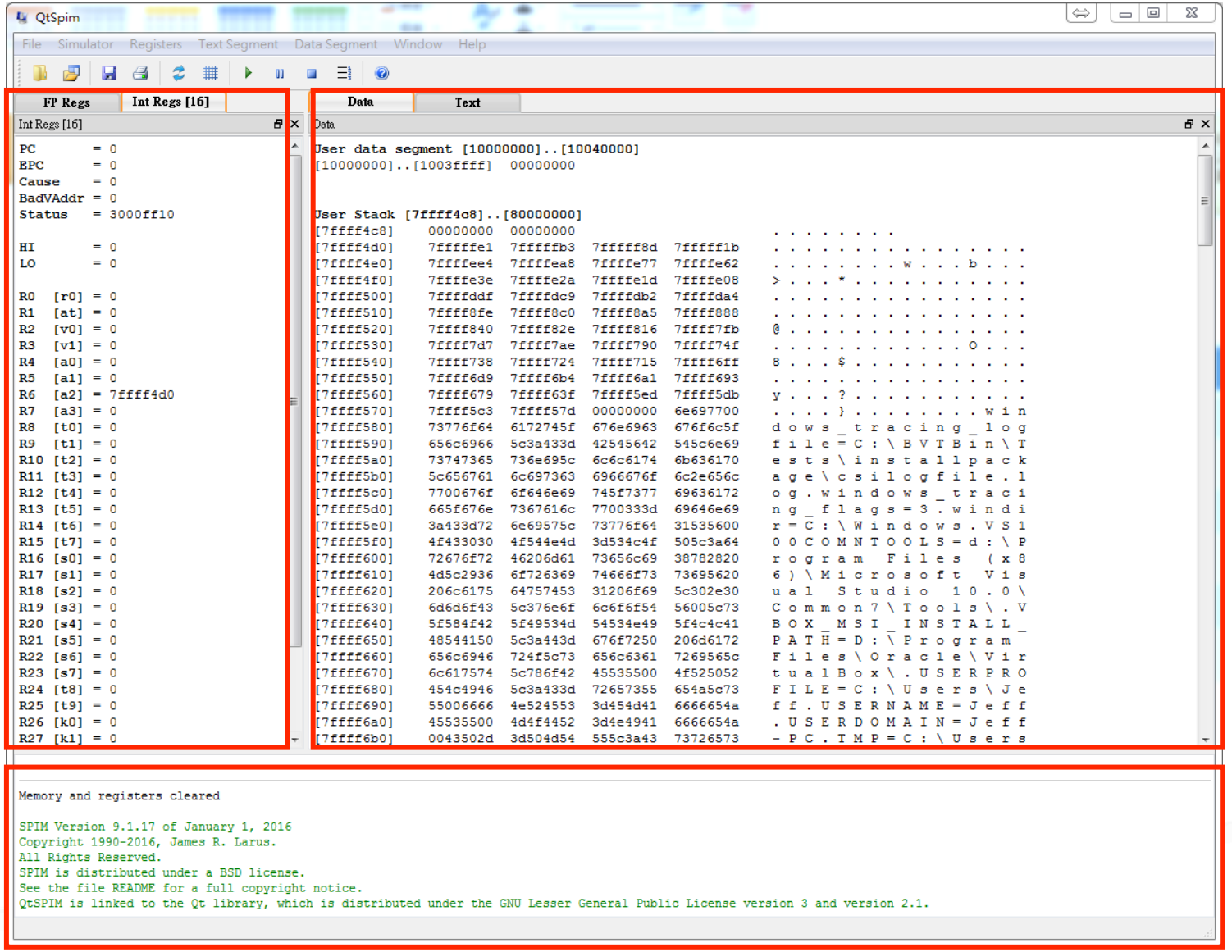
- http://alumni.cs.ucr.edu/~lgao/teaching/bison.html
- http://www.gnu.org/software/bison/manual/bison.html

Introduction to SPIM

- SPIM: MIPS32 simulator
- Installation: http://pages.cs.wisc.edu/~larus/spim.html
  - Windows: PCSpim
  - Linux/MacOSX: spim/xspim

Environment

Registers

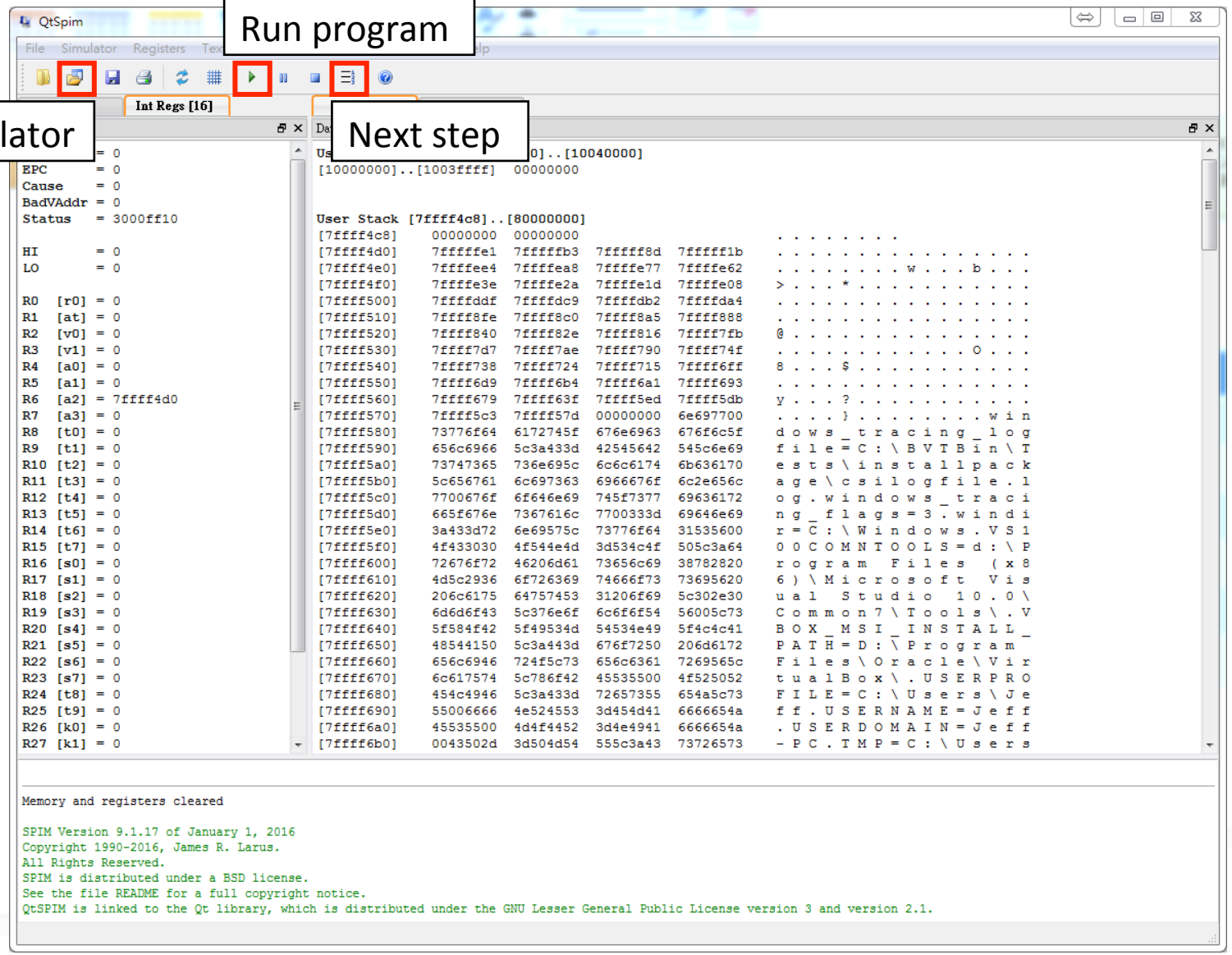


Text/Data Segment

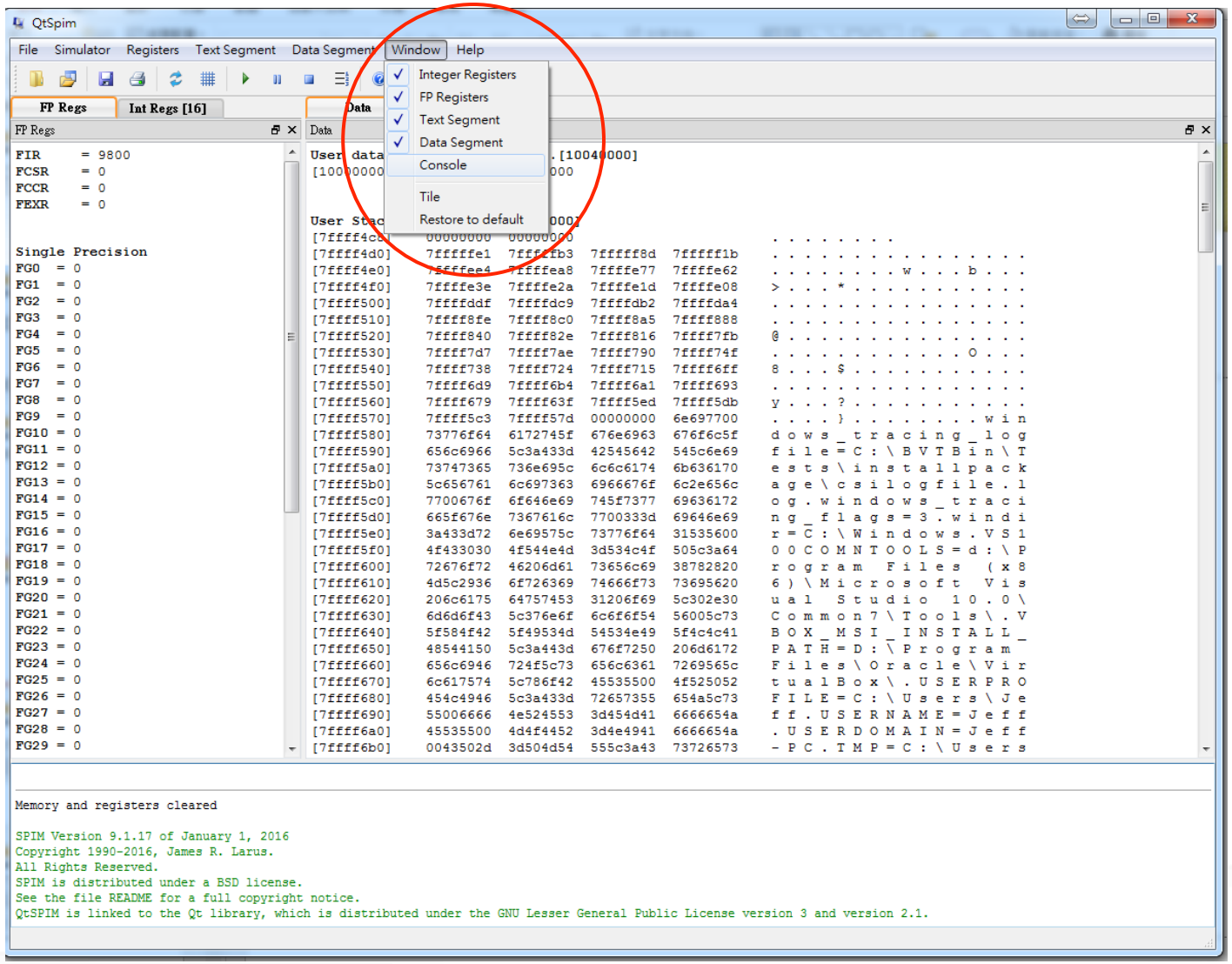
SPIM messages

Environment - Execution

Load asm and reset simulator



Environment - Enable Console



Registers

Number	Name	Usage	Number	Name	Usage
$0	$0	Constant 0	$24 - $25	$t8 - $t9	Temporary
$1	$at	Reserved for assembler	$26 - $27	$k0 - $k1	Reserved for OS kernel
$2 - $3	$v0 - $v1	Value returned by subroutines	$28	$gp	Pointer to global area
$4 - $7	$a0 - $a3	Subroutine Arguments	$29	$sp	Stack pointer
$8 - $15	$t0 - $t7	Temporary(not preserved across call)	$30	$fp	Frame pointer
$16 - $23	$s0 - $s7	Saved temporary(preserved across call)	$31	$ra	Return address

Assembler Directives

Directives	Usage
.asciiz str	Store the string in memory and null-terminate it
.byte b1, ..., bn	Store the n values in successive bytes of memory
.data	The following data items should be store in the data segment
.globl sym	Declare that symbol sym is global and can be referenced from other files
.half h1, ..., hn	Store the n values in successive memory halfwords
.space n	Allocate n bytes of space in the current segment(must be data segment)
.text	The next items are put in the user text segment
.word w1, ..., wn	Store the n 32-bit quantities in successive memory words

- data and space directive are useful for heap manipulation

System Services

Service	System Call Code($v0)	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		

System Service Example

- L4~L5: read a integer from console into $v0
- L6: move the value from $v0 to $a0
- L7~L8: print a integer in $a0
- L9~L10: stop program

Filename: service.s

```
1       .text
2       .globl main
3   main:
4       li   $v0, 5
5       syscall
6       move $a0, $v0
7       li   $v0, 1
8       syscall
9       li   $v0, 10
10      syscall
```

Instruction Set - Arithmetic

Operation	Destination	Source 1	Source 2	Mnemonic	Remark
add	Rdest,	Rsrc1,	Src2	Rdest = Rsrc1 + Src2	Add
div	Rdest,	Rsrc1,	Src2	Rdest = Rsrc1 / Src2	Divide
mul	Rdest,	Rsrc1,	Src2	Rdest = Rsrc1 * Src2	Multiply
neg	Rdest,	Rsrc		Rdest = -Rsrc	Negate Value
not	Rdest,	Rsrc		Rdest = !Rsrc	
sub	Rdest,	Rsrc1,	Src2	Rdest = Rsrc1 - Src2	Subtract
li	Rdest,	imm		Rdest = imm	Load immediate
move	Rdest,	Rsrc		Rdest = Rsrc	Move

- Src2 can either be a register or an immediate value(a 16 bit integer)
- All instructions with overflow, expects multiply operation

Arithmetic Example

- jal label - jump and link
  - jump to label and update return address
- b label - branch
  - jump to label
- print - function
  - print $a0 and newline
- exit - function
  - stop program

Filename: arithmetic.s

```
1       .text
2       .globl main
3   main:
4       li   $t0, 0
5       move $a0, $t0
6       jal print          → 0
7       add  $t0, $t0, 1
8       move $a0, $t0
9       jal print          → 1
10      sub  $t0, $t0, 2
11      move $a0, $t0
12      jal print          → -1
13      mul  $t0, $t0, -2
14      move $a0, $t0
15      jal print          → 2
16      div  $t0, $t0, -1
17      move $a0, $t0
18      jal print          → -2
19      b exit
```

Instruction Set - Comparison

Operation	Destination	Source 1	Source 2	Mnemonic	Remark
seq	Rdest,	Rsrc1,	Src2	Rdest = (Rsrc1 == Src2 ? 1 : 0)	Equal
sge	Rdest,	Rsrc1,	Src2	Rdest = (Rsrc1 >= Src2 ? 1 : 0)	Greater Than Equal
sgt	Rdest,	Rsrc1,	Src2	Rdest = (Rsrc1 > Src2 ? 1 : 0)	Greater Than
sle	Rdest,	Rsrc1,	Src2	Rdest = (Rsrc1 <= Src2 ? 1 : 0)	Less Than Equal
slt	Rdest,	Rsrc1,	Src2	Rdest = (Rsrc1 < Src2 ? 1 : 0)	Less Than
sne	Rdest,	Rsrc1,	Src2	Rdest = (Rsrc1 != Src2 ? 1 : 0)	Not Equal

- Src2 can either be a register or an immediate value(a 16 bit integer)

Comparison Example

Filename: comparison.s

```
1       .text
2       .globl main
3   main:
4       li   $t0, 0
5       li   $t1, 1
6       seq  $a0, $t0, $t1
7       jal print   ⇒ 0
8       sge  $a0, $t0, $t1
9       jal print   ⇒ 0
10      sgt  $a0, $t0, $t1
11      jal print   ⇒ 0
12      sle  $a0, $t0, $t1
13      jal print   ⇒ 1
14      slt  $a0, $t0, $t1
15      jal print   ⇒ 1
16      sne  $a0, $t0, $t1
17      jal print   ⇒ 1
18      b exit
```

Instruction Set - Memory Access

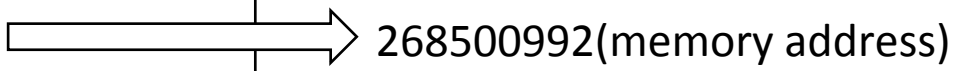
Operation	Register	Address	Mnemonic	Remark
la	Rdest	address	Rdest = address	Load Address
lw	Rdest	address	Rdest = Mem[address]	Load Word
sw	Rsrc	address	Mem[address] = Rsrc	Store Word

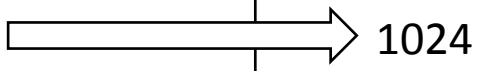
- There are different data type you can manipulate
  - e.g. byte, half-word, double-word
  - operation with suffix “b”, “h”, “d”, e.g. lb, lh, sd, etc...

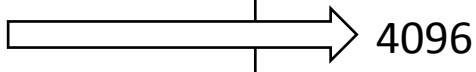
Memory Access Example

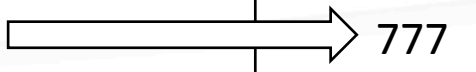
Filename: memory.s

```
1       .data
2   varA: .word 1024
3   varB: .word 4096
4
5       .text
6       .globl main
7   main:
8       la  $a0, varA
9       jal print
10      lw  $a0, varA
11      jal print
12      la  $a0, varB
13      jal print
14      li  $t0, 777
15      sw  $t0, varB
16      lw  $a0, varB
17      jal print
18      b exit
```









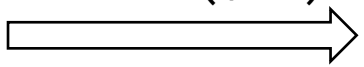
Addressing Modes

Format	Address Computation
(register)	contents of register
imm	immediate
imm (register)	immediate + contents of register
symbol	address of symbol
symbol imm	address of symbol + or - immediate
symbol imm (register)	address of symbol + or - (immediate + contents of register)

Address Modes Example

Filename: addressing.s

```
1       .data
2   varA: .word 1024
3   varB: .word 4096
4   varC: .word 512
5
6       .text
7       .globl main
8   main:
9       li  $t0, 4
10      lw  $a0, varA + 4($t0)
11      jal print
12      b exit
```



512

Instruction Set - Branch and Jump

Operation	Source 1	Source 2	Destination	Mnemonic	Remark
b			label	Branch to label	Branch
beq	Rsrc1,	Src2,	labal	Branch to label if Rsrc1 == Src2	Branch on Equal
bge	Rsrc1,	Src2,	labal	Branch to label if Rsrc1 >= Src2	Branch on GTE
bgt	Rsrc1,	Src2,	labal	Branch to label if Rsrc1 > Src2	Branch on GT
ble	Rsrc1,	Src2,	labal	Branch to label if Rsrc1 <= Src2	Branch on LTE
blt	Rsrc1,	Src2,	labal	Branch to label if Rsrc1 < Src2	Branch on LT
bne	Rsrc1,	Src2,	labal	Branch to label if Rsrc1 != Src2	Branch on NEQ
jal			label	Branch to label/Rsrc and save address of next instruction in $ra	Branch and Link
jalr			Rsrc		

- jal/jalr are used in calling subroutine
- There are zero comparison branch, compare to zero instead of Src2
  - operation suffix with 'z', e.g. beqz, bgez, bgtz

Branch and Jump Example

Filename: branch.c

```
1   int main {
2       int x, y;
3       char *strEq = “Equal”;
4       char *strNe = “Not equal”;
5       char *output;
6
7       scanf(“%d”, &x)
8       scanf(“%d”, &y)
9
10      if(x == y)
11          output = strEq;
12      else
13          output = strNe;
14
15      printf(“%s”, output);
16
17      return 0;
18  }
```

Filename: branch.s

```
1       .data
2   strEq: asciiz “Equal”
3   strNe: asciiz “Not equal”
4       .text
5       .globl main
6   main:
7       jal   read
8       move $t0, $v0
9       jal   read
10      move $t1, $v0
11      beq   $t0, $t1, printEq
12  printNe:
13      la    $a0, strNe
14      b     endif
15  printEq:
16      la    $a0, strEq
17      b     endif
18  endif:
19      jal   print
20      b     exit
```

Function Example

- jal will store the address of the next instruction
  - e.g. L6
- L11 can jump back to L6
  - address of L6 is stored in $ra
  - branch to $ra directly

Filename: function.s

```
1       .text
2       .globl main
3   main:
4       li  $a0, 123
5       jal print
6       b exit
7
8   print:
9       li  $v0, 1
10      syscall
11      b   $ra
12
13  exit:
14      li  $v0, 10
15      syscall
```

Reference

- https://www.cs.tcd.ie/John.Waldron/itral/spim_ref.html#directives
- http://courses.cs.washington.edu/courses/cse410/08sp/notes/spim/SpimTutorial.pdf
- http://students.cs.tamu.edu/tanzir/csce350/reference/syscalls.html
- http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html
- http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm