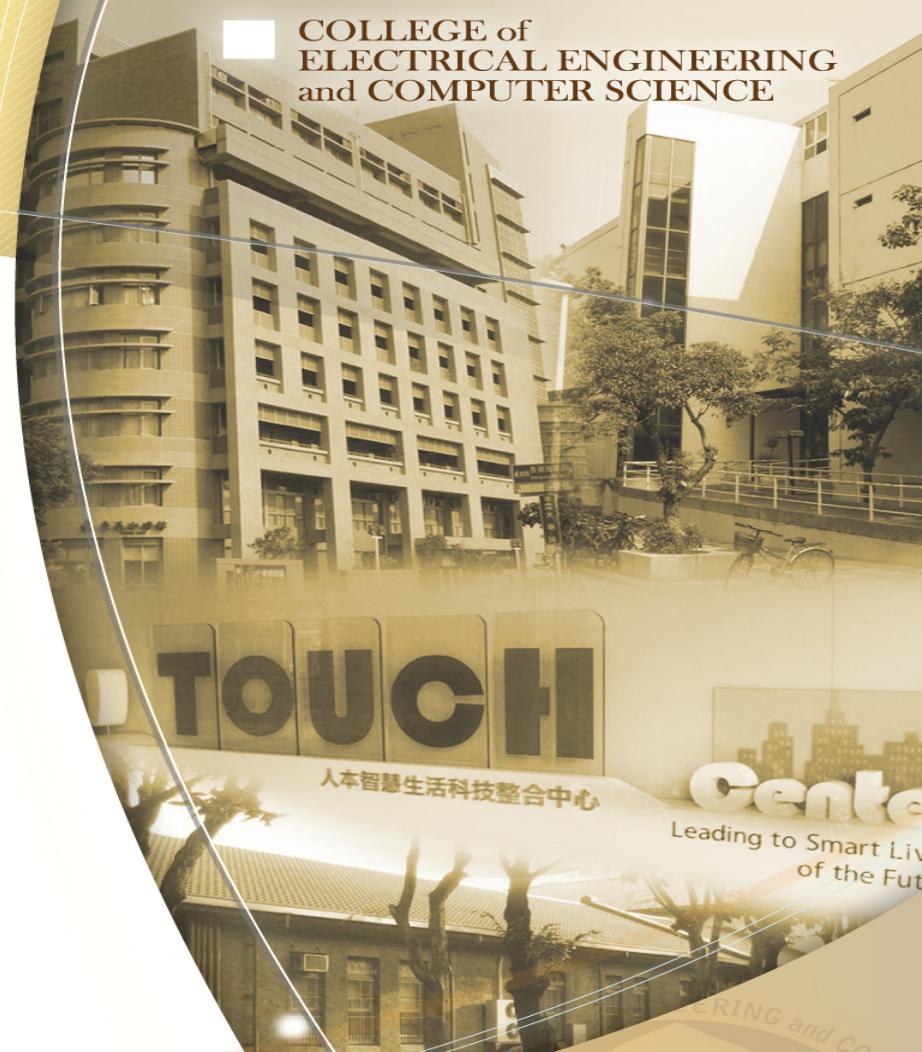


# Compiler Course Program Assignment

Speaker: Jeff Liaw



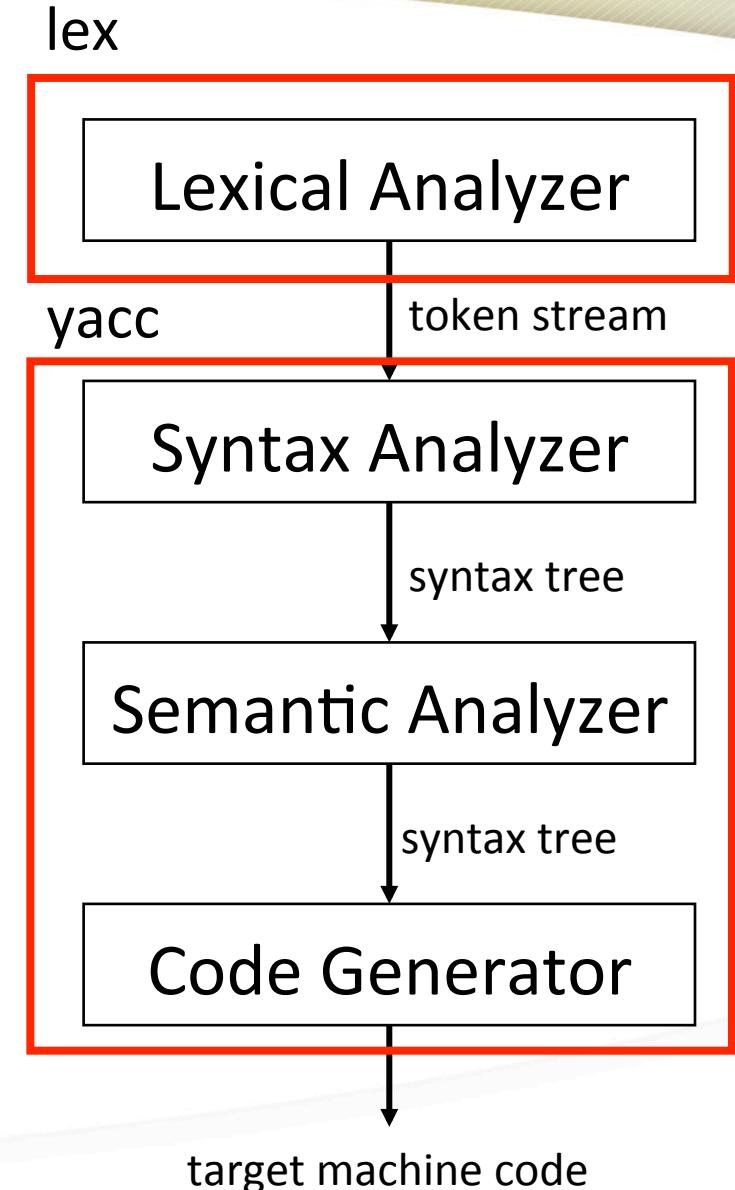


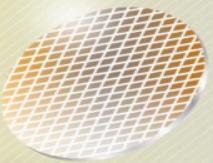
# Program Assignment

- 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

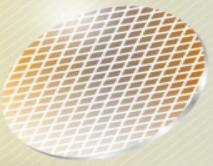
■= ! + - \* /

## ●Comparison Operators

■== != < > <= >= && ||

## ●Special Symbols

■[ ] ( ) { } ; ,



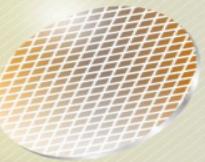
# Token Format(2)

## ● 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)

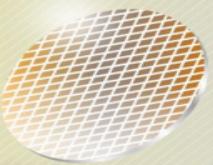
## ● Char

- 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



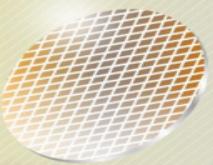
# Grammar

## ● 參考附件grammar.txt

head(left side) → Program  
body(right side) → VarDeclList FunDeclList  
VarDeclList → VarDecl1 VarDeclList | epsilon  
VarDecl1 → Type id ; | Type id [ num ] ;  
production 1 → Type id ;  
production 2 → Type id [ num ] ;  
FunDeclList → FunDecl1 | FunDecl FunDeclList  
FunDecl1 → FunDecl1 FunDeclList  
FunDecl → FunDecl1  
Type id ( ParamDeclList ) Block

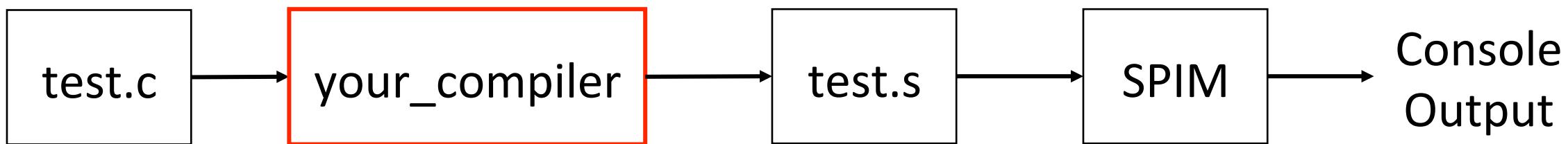


Program → VarDeclList FunDeclList  
VarDeclList → VarDecl1 VarDeclList | epsilon  
VarDecl1 → Type id ; | Type id [ num ] ;  
FunDeclList → FunDecl1 | FunDecl FunDeclList  
FunDecl1 → Type id ( ParamDeclList ) Block



# Expected Result

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

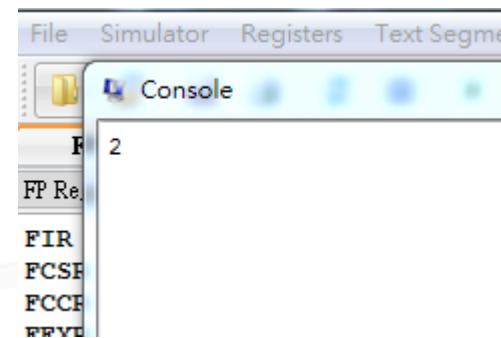


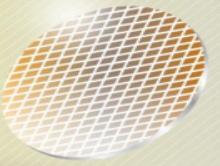
```
int main() {  
    int x;  
    x = 2;  
    print x;  
}
```

your\_compiler

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

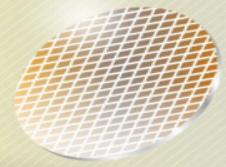
SPIM





# Grading

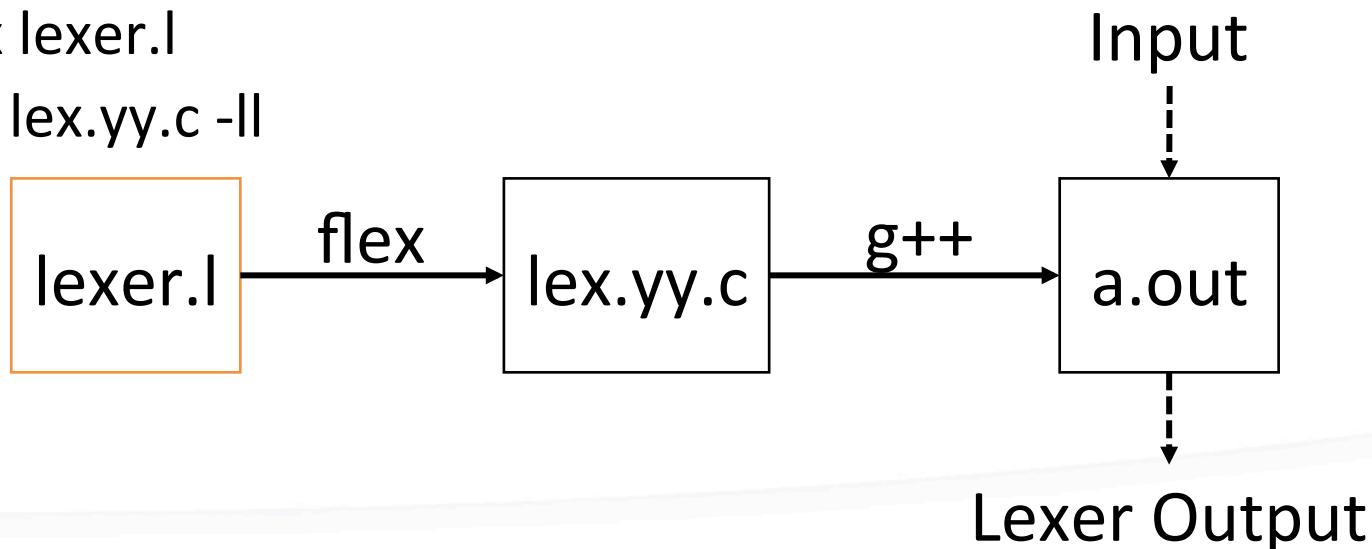
- IO: read/print - 10%
- Arithmetic - 10%
  - priority - 5%
- Conditional - 15%
- Loop - 15%
- Array - 15%
- Function call - 20%
- Demo - 10%

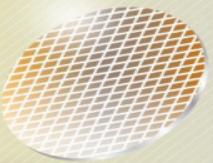


# Introduction to Flex

- 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 -lI



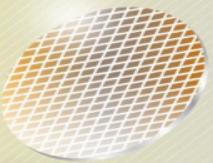


# Rule Files

- 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
```

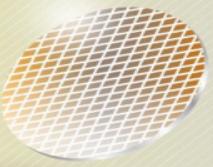


# Lex Regular Expression

x	the character “x”	x\$	an x at the end of a line
“x”	an “x”, even if x is an operator	x?	an optional x
\x	an “x”, even if x is an operator	x*	0,1,2, ... instances of x
[xy]	the character x or y	x+	1,2,3, ... instances of x
[x-z]	the characters x, y or z	x y	an x or an y
[^x]	any character but x	(x)	an x
.	any character but newline	x/y	an x but only if followed by y
^x	an x at the beginning of a line	{xx}	the translation of xx from the definitions section
<y>x	an x When Lex is in start condition y	x{m,n}	m through n occurrences of x

## Operators

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



# Lex Actions

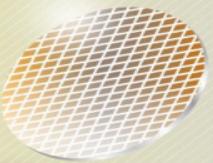
- Rule specifies an action to perform if input matches the regular expression or definition
- The action is specified by writing regular C code
- The default action is copying input to output

● e.g.

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

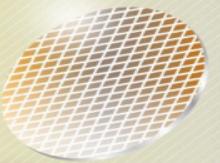
## Rule File Structure

```
%{  
Declarations  
%}  
  
Definitions  
%%  
Rules  
%%  
User subroutines
```



# Lex Predefined Variables

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



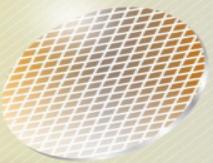
# Calculator Example

- 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	void yyerror();
3	%}

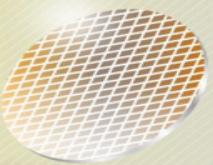
## Rule File Structure

%{  
**Declarations**  
%}

Definitions  
%%

Rules  
%%

User subroutines



# Definitions

## ● Name two useful regular expression

- digit: [0-9]
- int\_const: {digit}+

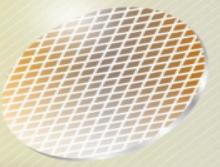
Code Section	
1	digit [0-9]
2	int_const {digit}+

## Rule File Structure

```
%{  
Declarations  
%}
```

## Definitions

```
%%  
Rules  
%%  
User subroutines
```



# Rules

- Print some message during lexical analyzing
  - integer\_const, plus and multiply

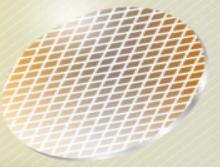
## Code Section

```
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

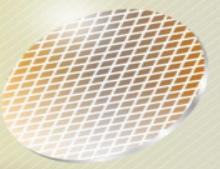
```
1 int main() {
2     yylex();
3     return 0;
4 }
5
6 void yyerror() {
7     extern int yylineno;
8     extern char *yytext;
9
10    fprintf(stderr, "Error: at symbol '\"%s\'' on line %d\n"
11            , yytext, yylineno);
12 }
```

### Rule File Structure

%{  
Declarations  
%}

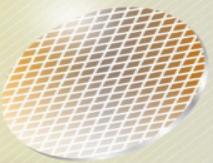
Definitions  
%%  
Rules  
%%

User subroutines



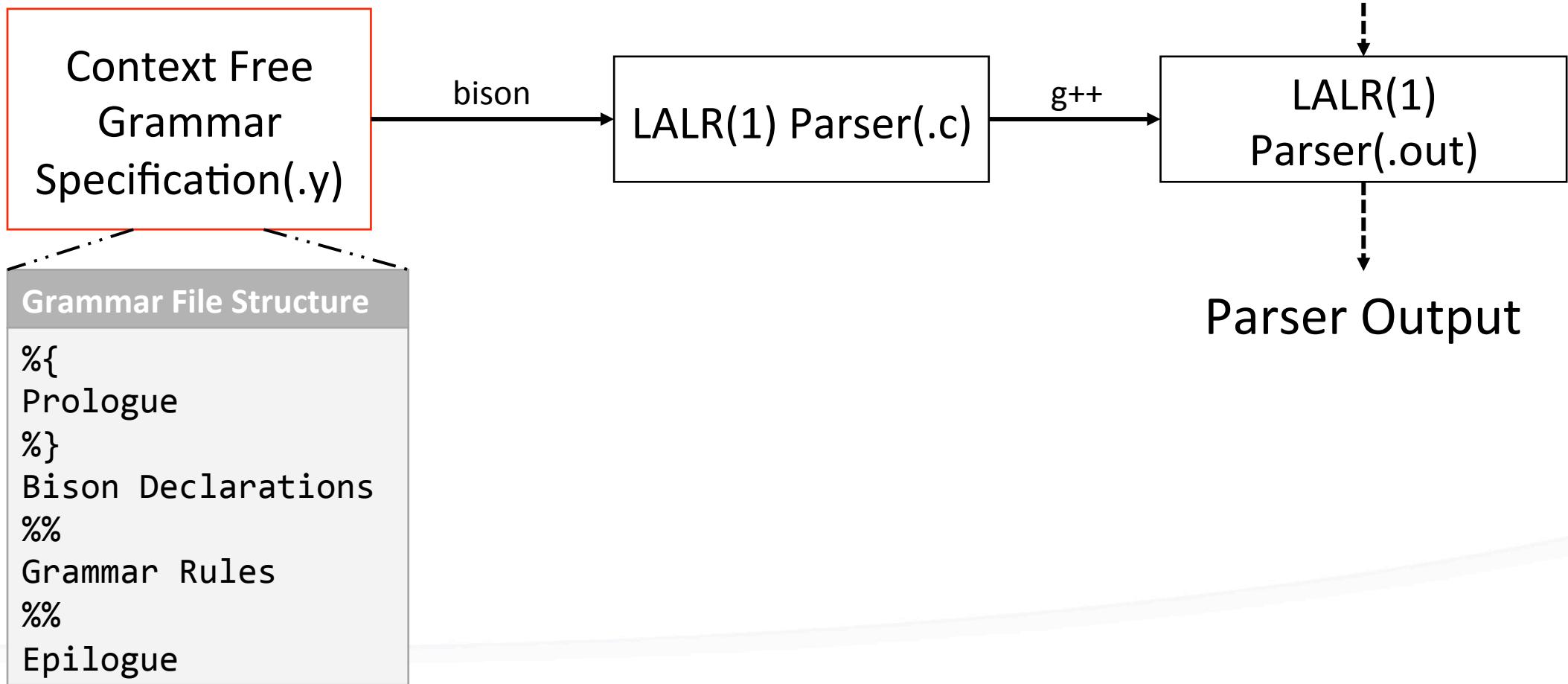
# Reference

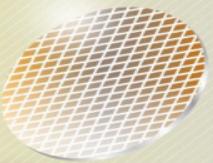
- <http://flex.sourceforge.net/#overview>
- <http://dinosaur.compilertools.net/lex/index.html>



# Introduction to Bison

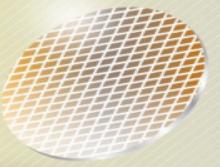
- LALR(1) parser generator under the GNU license





# 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

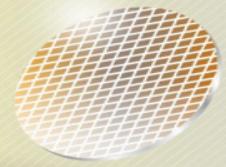
- Between “%{“ and “%}”
- Content between those is copied verbatim into output file
- Declare functions prototype, so we can use later

## Code Section

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

## Grammar File Structure

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

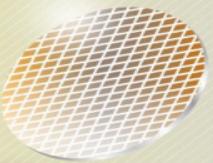


# Bison Declarations(1)

- 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
```



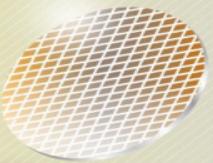
# Bison Declarations(2)

## 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

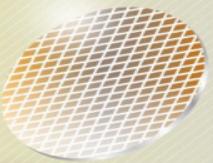
- e.g. exp: exp '+' exp { printf("sum two numbers\n"); }

- 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

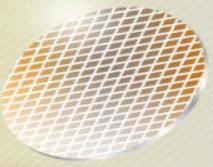
- value of component n is \$n
- modifiable lvalue is \$\$

### 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

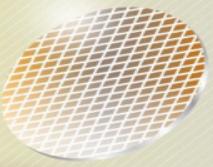
- Copy verbatim to the end of the output file
- Put your subroutine here

## Code Section

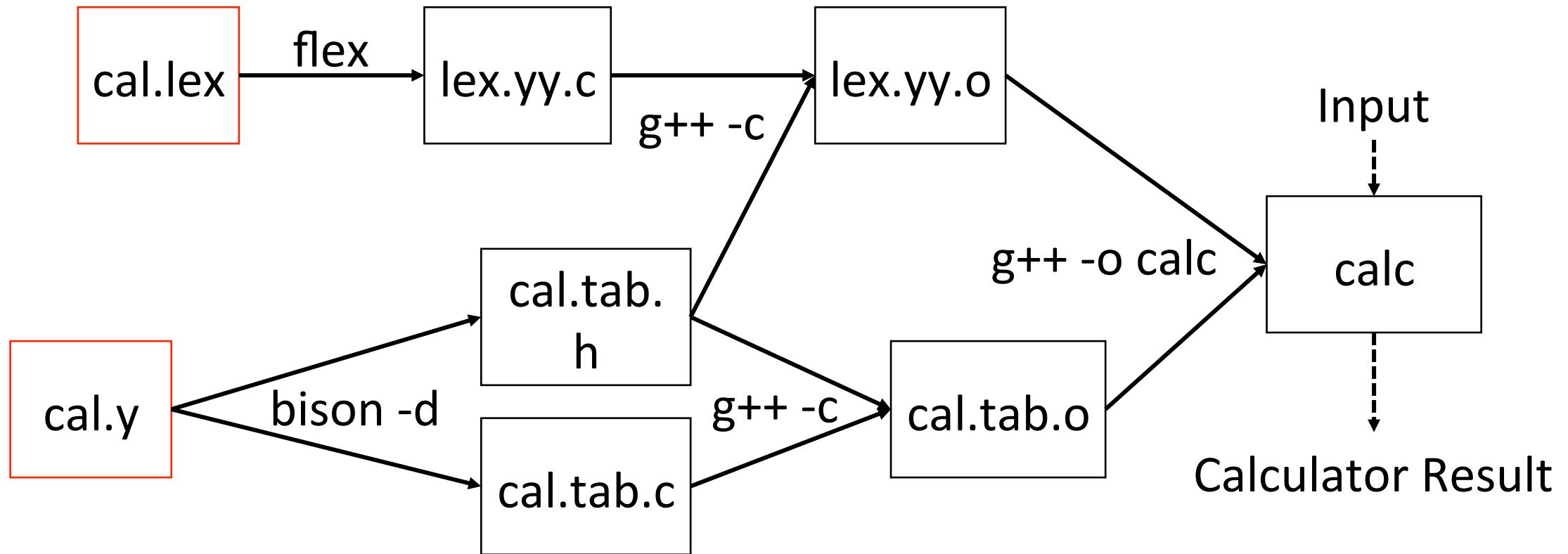
```
1 int main() {  
2     yyparse();  
3  
4     return 0;  
5 }
```

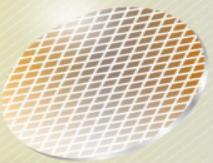
## Grammar File Structure

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



# Combine Flex and Bison





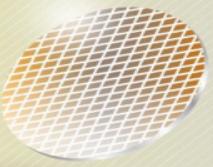
# Pass Token from Flex to Bison

- Recall lex predefined variables
  - `yylval`

Filename: cal.tab.h

```
55 #if ! defined YYSTYPE && ! defined YYSTYPE_IS_DEFINED
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_DEFINED 1
67 # define YYSTYPE_IS_TRIVIAL 1
68 #endif
69
70 extern YYSTYPE yylval;
```

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

Code Section: calc.y(grammar rules segment)

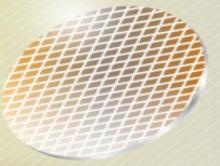
```
1 exp: INTEGER_LITERAL { $$ = $1; }
2   | exp PLUS exp { $$ = $1 + $3; }
3   | exp MULT exp { $$ = $1 * $3; }
```

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 }
```

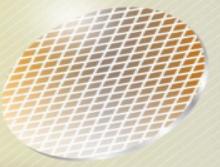
Filename: cal.tab.h

```
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
```



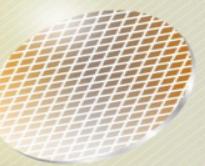
# Reference

- <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

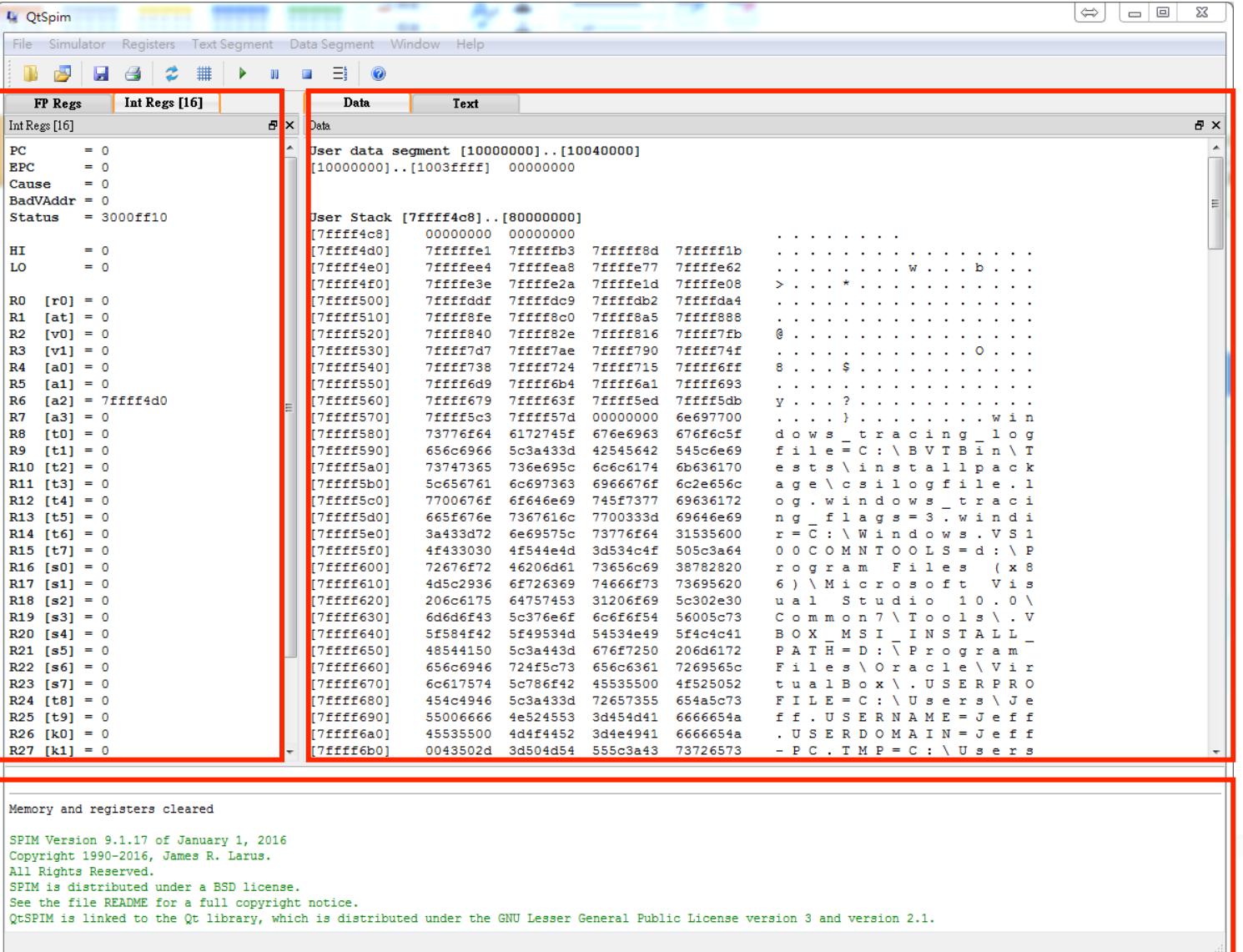


# Environment

**Registers**

**Text/Data Segment**

**SPIM messages**



The screenshot shows the QtSpim debugger interface. The top menu bar includes File, Simulator, Registers, Text Segment, Data Segment, Window, and Help. The main window has three tabs: FP Regs, Int Regs [16], and Data. The Int Regs [16] tab is selected, displaying various processor registers like PC, EPC, Cause, BadVAddr, Status, HI, LO, and multiple R0-R27 registers. The Data tab shows memory dump sections for User data segment and User Stack, with addresses ranging from 10000000 to 7ffff6b0. The bottom status bar displays SPIM version information and copyright notice.

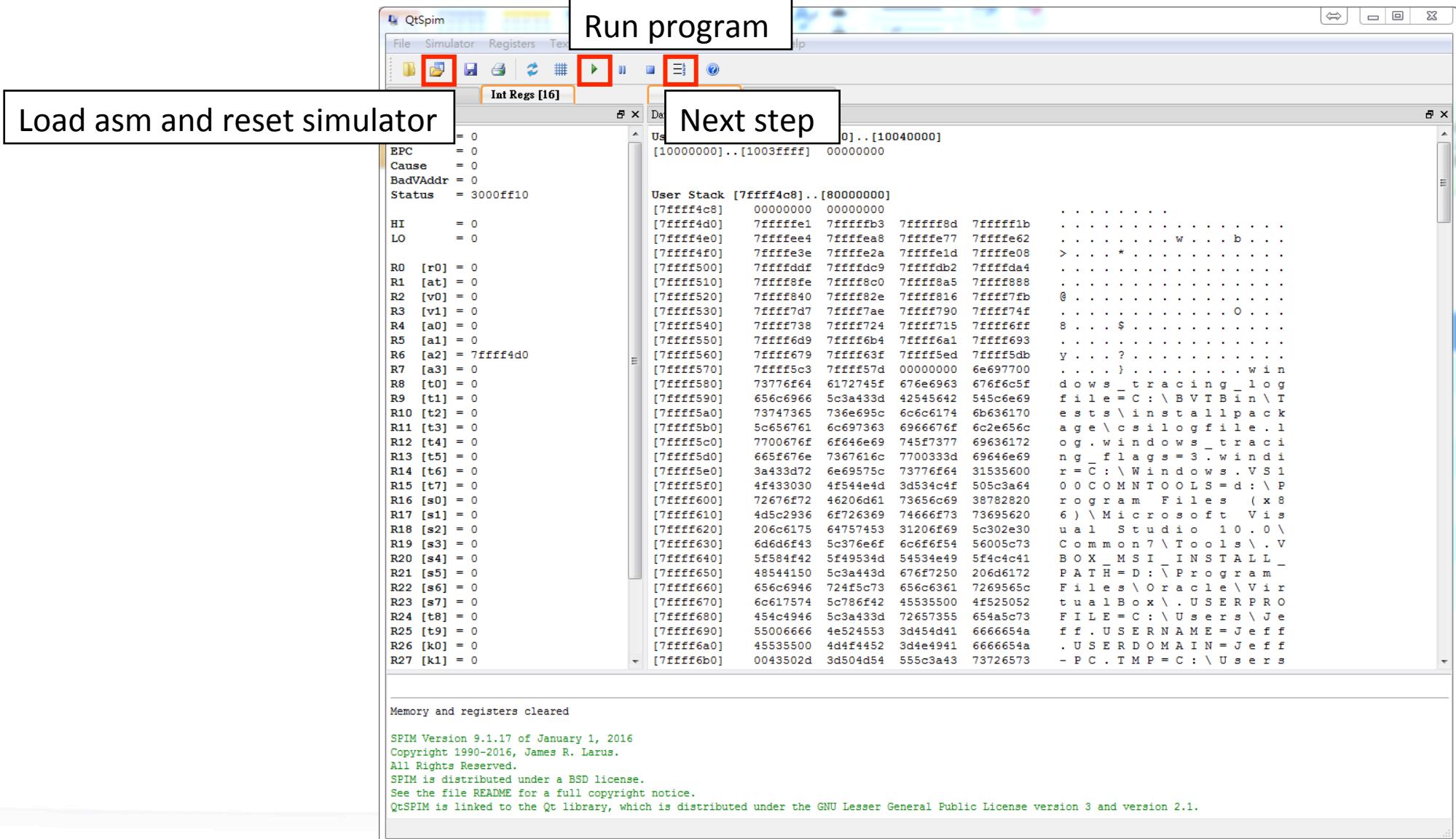
```

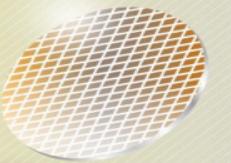
Registers
Text/Data Segment
SPIM messages

QtSpim
File Simulator Registers Text Segment Data Segment Window Help
FP Regs Int Regs [16]
Int Regs [16]
PC = 0
EPC = 0
Cause = 0
BadVAddr = 0
Status = 3000ff10
HI = 0
LO = 0
R0 [r0] = 0
R1 [at] = 0
R2 [v0] = 0
R3 [v1] = 0
R4 [a0] = 0
R5 [a1] = 0
R6 [a2] = 7ffff4d0
R7 [a3] = 0
R8 [t0] = 0
R9 [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
Data Text
User data segment [10000000]..[10040000]
[10000000]..[1003ffff] 00000000
User Stack [7ffff4c8]..[80000000]
[7ffff4c8] 00000000 00000000
[7ffff4d0] 7fffffe1 7fffffb3 7fffff8d 7fffff1b . . . . .
[7ffff4e0] 7ffffee4 7fffffea8 7ffffe77 7fffffe62 . . . . w . b . .
[7ffff4f0] 7fffffe3e 7fffffe2a 7fffffeid 7fffffe08 > . . . * . . . . .
[7ffff500] 7fffffdcf 7fffffdcb2 7fffffdab 7fffffd4 . . . . .
[7ffff510] 7ffff8fe 7ffff8c0 7ffff8a5 7ffff888 . . . . .
[7ffff520] 7ffff840 7ffff82e 7ffff816 7ffff7fb @ . . . . .
[7ffff530] 7ffff7d7 7ffff7ae 7ffff790 7ffff74f . . . . o . .
[7ffff540] 7ffff738 7ffff724 7ffff715 7ffff6ff 8 . . . $ . . . . .
[7ffff550] 7ffff6d9 7ffff6b4 7ffff6a1 7ffff693 . . . . .
[7ffff560] 7ffff679 7ffff63f 7ffff5ed 7ffff5db y . . . ? . . . . .
[7ffff570] 7ffff5c3 7ffff57d 00000000 6e697700 . . . } . . . . w i n
[7ffff580] 73776f64 6172745f 676e6963 676f6c5f d o w s _ t r a c i n g _ l o g
[7ffff590] 656cc6966 5c3a433d 42545642 545c6e69 f i l e = C : \ B V T B i n \ T
[7ffff5a0] 73747365 736e695c 6c6c6174 6b636170 e s t s \ i n s t a l l p a c k
[7ffff5b0] 5c656761 6c697363 6966676f 6c2e656c a g e \ c s i l o g f i l e . 1
[7ffff5c0] 7700676f 6f646e69 745f7377 69636172 o g . w i n d o w s _ t r a c i
[7ffff5d0] 665f676e 7367616c 7700333d 69646e69 n g _ f l a g s = 3 _ w i n d i
[7ffff5e0] 3a433d72 6e69575c 73776f64 31535600 r = C : \ W i n d o w s . V S 1
[7ffff5f0] 4f433030 4f544e4d 3d534c4f 505c3a64 0 0 C O M N T O O L S = d : \ P
[7ffff600] 72676f72 46206d61 73656c69 38782820 r o g r a m _ F i l e s ( x 8
[7ffff610] 4d5c2936 6f726369 74666f73 73695620 6 ) \ M i c r o s o f t _ V i s
[7ffff620] 206c6175 64757453 31206f69 5c302e30 u a l _ S t u d i o _ 1 0 . 0 \
[7ffff630] 6d6d6f43 5c376e6f 6c6f6f54 56005c73 C o m m o n 7 \ T o o l s \ . V
[7ffff640] 5f584f42 5f49534d 54534e49 5f4c4c41 B O X _ M S I _ I N S T A L L _ -
[7ffff650] 48544150 5c3a433d 676f7250 206d6172 P A T H = D : \ P r o g r a m
[7ffff660] 656cc6946 724f5c73 656c6361 7269565c F i l e s \ O r a c l e \ V i r
[7ffff670] 6c617574 5c786f42 45535500 4f525052 t u a l _ B o x \ . U S E R P R O
[7ffff680] 454c4946 5c3a433d 72657355 654a5c73 F I L E = C : \ U s e r s \ J e
[7ffff690] 55006666 4e524553 3d454d41 6666654a f f . U S E R N A M E = J e f f
[7ffff6a0] 45535500 4d4f4452 3d4e4941 6666654a . U S E R D O M A I N = J e f f
[7ffff6b0] 0043502d 3d504d54 555c3a43 73726573 - P C . T M P = C : \ U s e r s
Memory and registers cleared
SPIM Version 9.1.17 of January 1, 2016
Copyright 1990-2016, James R. Larus.
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
QtSPIM is linked to the Qt library, which is distributed under the GNU Lesser General Public License version 3 and version 2.1.

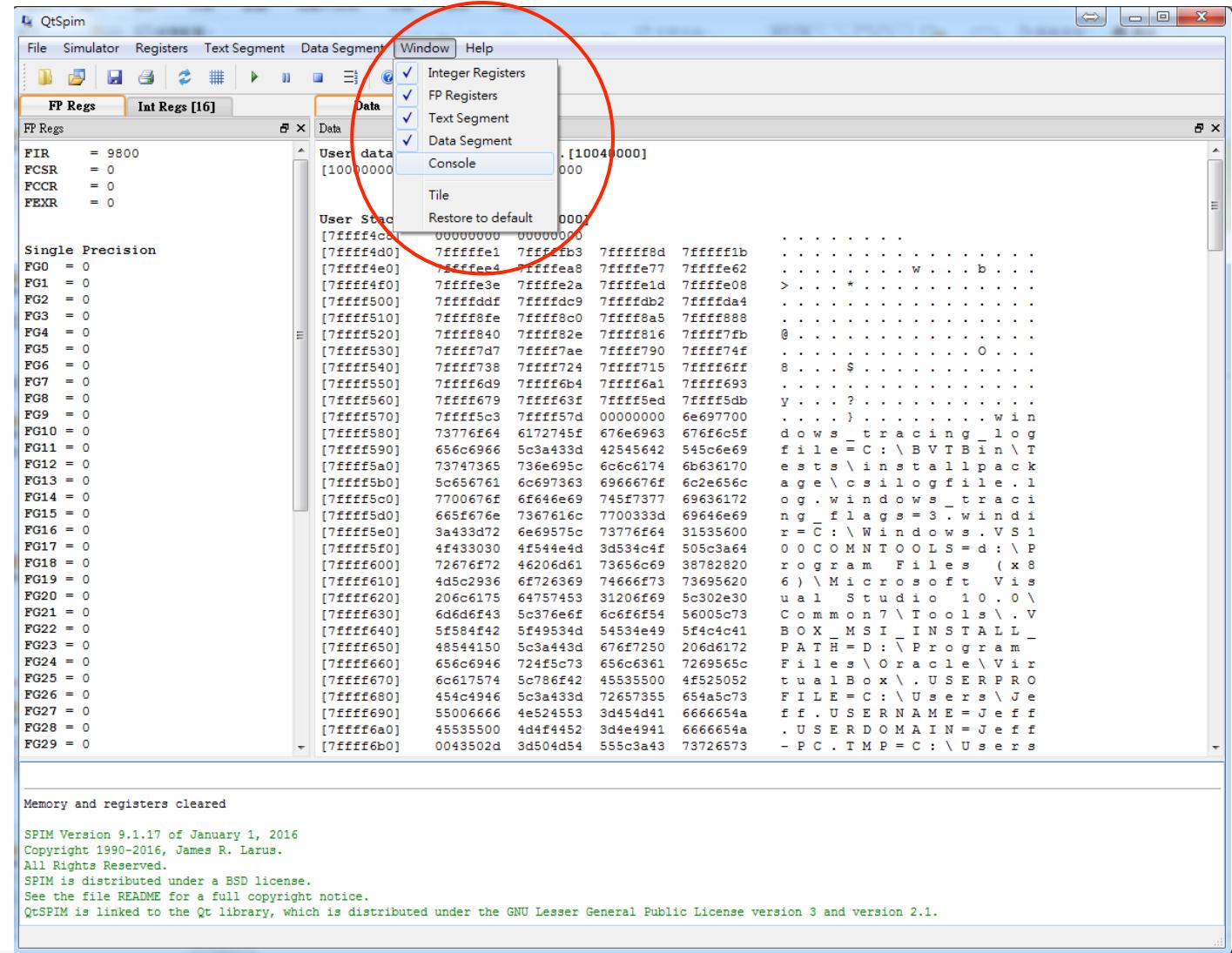
```

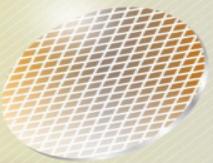
# Environment - Execution





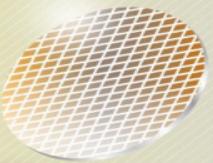
# Environment - Enable Console





# 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		

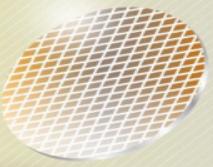


# System Service Example

- 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

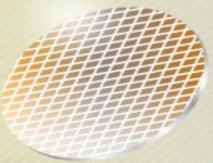
- Src2 can either be a register or an immediate value(a 16 bit integer)
- All instructions with overflow, expects multiply operation



# 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

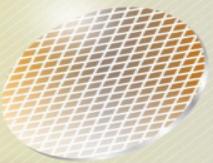
Filename: arithmetic.s	
1	.text
2	.globl main
3	main:
4	li \$t0, 0
5	move \$a0, \$t0
6	jal print
7	add \$t0, \$t0, 1
8	move \$a0, \$t0
9	jal print
10	sub \$t0, \$t0, 2
11	move \$a0, \$t0
12	jal print
13	mul \$t0, \$t0, -2
14	move \$a0, \$t0
15	jal print
16	div \$t0, \$t0, -1
17	move \$a0, \$t0
18	jal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



# Instruction Set - Comparison

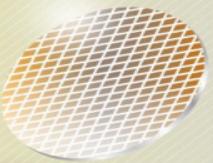
Operation	Destination	Source 1	Source 2	Mnemonic	Remark
seq	Rdest,	Rsrc1,	Src2	Rdest = (Rsrc1 == Src2 ? 1 : 0)	Equal
sge	Rdest,	Rsrc1,	Src2	Rdest = (Rsrc1 >= Src2 ? 1 : 0)	Greater Than Equal
sgt	Rdest,	Rsrc1,	Src2	Rdest = (Rsrc1 > Src2 ? 1 : 0)	Greater Than
sle	Rdest,	Rsrc1,	Src2	Rdest = (Rsrc1 <= Src2 ? 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

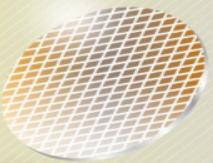
Filename: comparison.s	
1	.text
2	.globl main
3	main:
4	li    \$t0, 0
5	li    \$t1, 1
6	seq   \$a0, \$t0, \$t1
7	jal   print → 0
8	sge   \$a0, \$t0, \$t1
9	jal   print → 0
10	sgt   \$a0, \$t0, \$t1
11	jal   print → 0
12	sle   \$a0, \$t0, \$t1
13	jal   print → 1
14	slt   \$a0, \$t0, \$t1
15	jal   print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



# Instruction Set - Memory Access

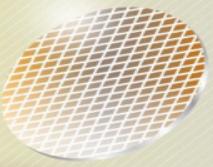
Operation	Register	Address	Mnemonic	Remark
la	Rdest	address	Rdest = address	Load Address
lw	Rdest	address	Rdest = Mem[address]	Load Word
sw	Rsrc	address	Mem[address] = Rsrc	Store Word

- 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...



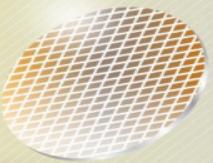
# Memory Access Example

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 → 268500992(memory address)
10	lw \$a0, varA
11	jal print → 1024
12	la \$a0, varB
13	jal print → 4096
14	li \$t0, 777
15	sw \$t0, varB
16	lw \$a0, varB
17	jal print → 777
18	b exit

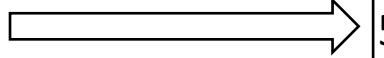


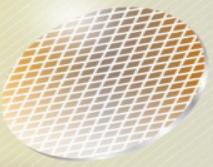
# Addressing Modes

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  512
12	b exit



# Instruction Set - Branch and Jump

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

- 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 }
```

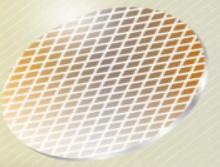
	Filename: branch.s
1	.data
2	strEq: .asciiz "Equal"
3	strNe: .asciiz "Not equal"
4	.text
5	.globl main
6	main:
7	jal read
8	move \$t0, \$v0
9	jal read
10	move \$t1, \$v0
11	beq \$t0, \$t1, printEq
12	printNe:
13	la \$a0, strNe
14	b endif
15	printEq:
16	la \$a0, strEq
17	b endif
18	endif:
19	jal print
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](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>