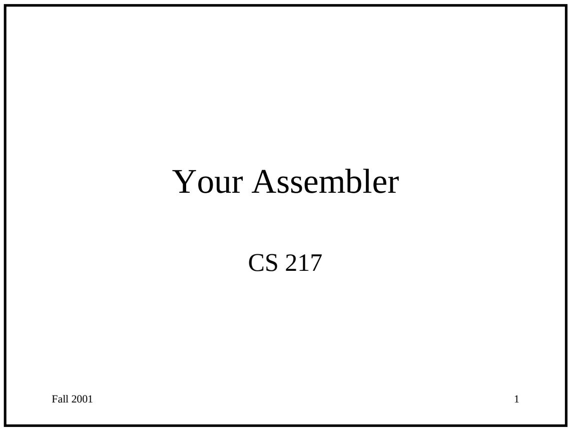
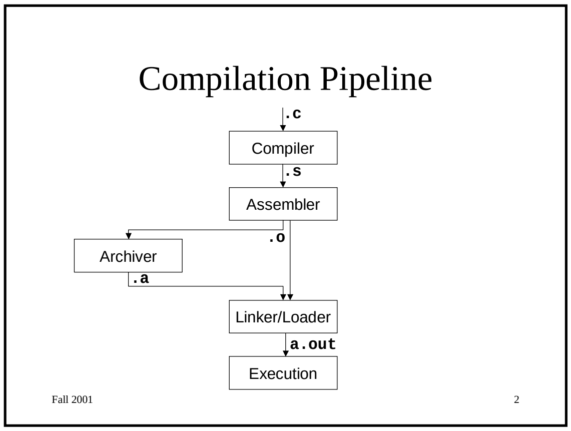
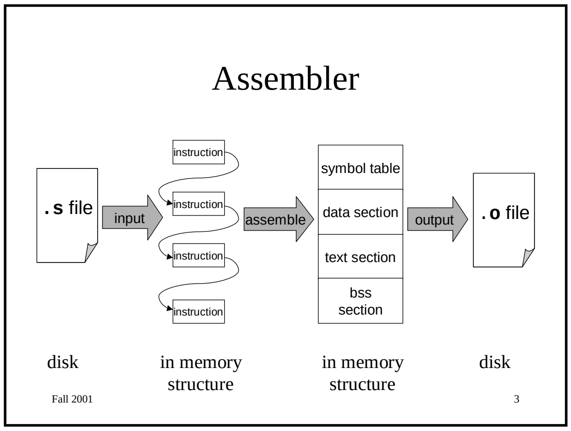


# Your Assembler

CS 217



## Compilation Pipeline



## Assembler

# Input Function

- Lexical Analyzer
  - group a stream of characters into tokens

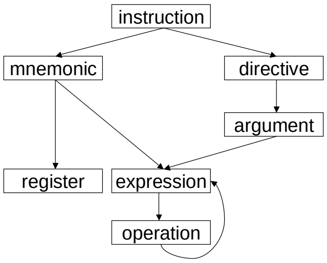
```
   add   “hello”  %r1     ,     10
<MNEMONIC><STR><REG><COMMA><INT>
```

- Syntactic Analyzer
  - check the syntax of the program

```
   instruction =
<MNEMONIC><REG><COMMA><REG><COMMA><REG>
```

- Instruction List Producer
  - produce an in-memory list of instruction data structures

# Input Data Structures



instruction

mnemonic

directive

argument

register

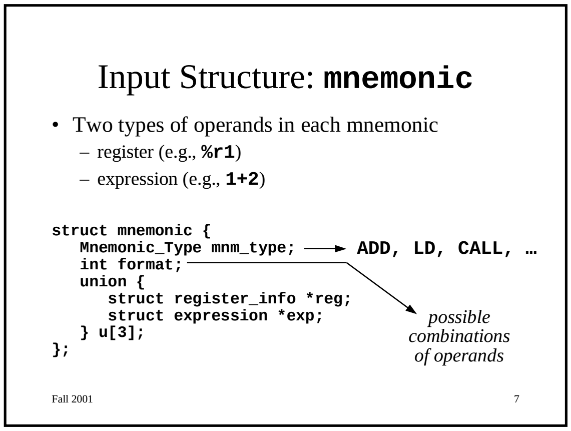
expression

operation

# Input Structure: `instruction`

- Three types of assembly instructions
  - label (symbol definition)
  - mnemonic (real or synthetic instruction)
  - directive (pseudo operation)

```
struct instruction {
   int instr_type; ───────────► LBL, MNM, DIR
   union {
      char *lbl;
      struct mnemonic *mnm;
      struct directive *dir;
   } u;
   struct instruction *next;
};
```



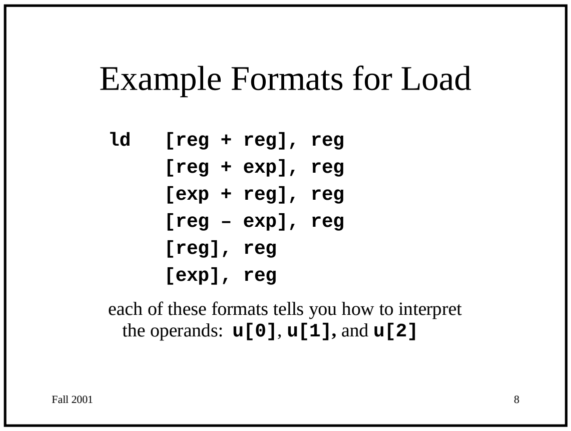
# Input Structure: **mnemonic**

- Two types of operands in each mnemonic
  - register (e.g., **%r1**)
  - expression (e.g., **1+2**)

```
struct mnemonic {
   Mnemonic_Type mnm_type;
   int format;
   union {
      struct register_info *reg;
      struct expression *exp;
   } u[3];
};
```

**ADD, LD, CALL, …**

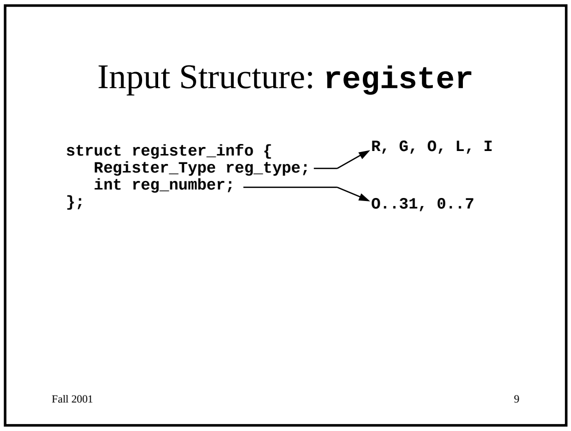
*possible combinations of operands*



# Example Formats for Load

```
ld   [reg + reg], reg
     [reg + exp], reg
     [exp + reg], reg
     [reg – exp], reg
     [reg], reg
     [exp], reg
```

each of these formats tells you how to interpret the operands: **u[0]**, **u[1]**, and **u[2]**



# Input Structure: **register**

```
struct register_info {
   Register_Type reg_type;
   int reg_number;
};
```

**R, G, O, L, I**

**0..31, 0..7**

# Input Structure: `expression`

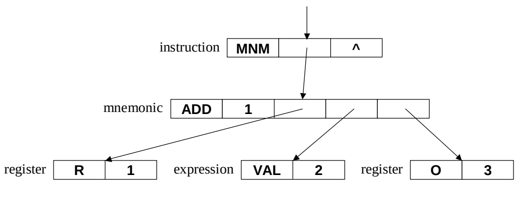
- Three types of expressions
  - symbol (e.g., **`loop`**)
  - integer (e.g., **`1`**)
  - operation (e.g., **`1+2`**)

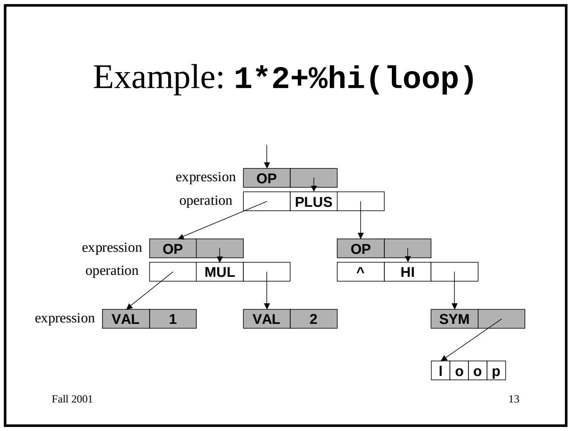
```
struct expression {
   int exp_type; ────────► SYM, VAL, OP
   union {
      char *sym;
      int val;
      struct operation *op;
   } u;
};
```

# Input Structure: `operation`

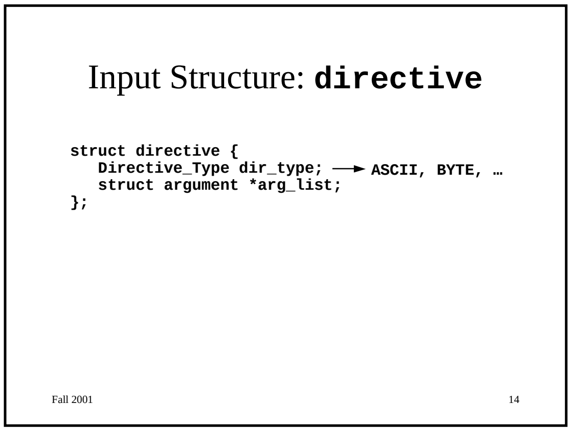
```
struct operation {
   Operation_Type op_type; ─────► PLUS, MUL, HI, …
   struct expression *left;
   struct expression *right;
};
```

# Example: `add %r1, 2, %o3`



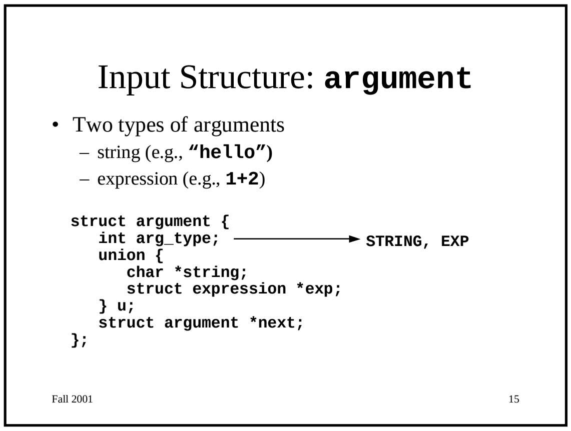


## Example: `1*2+%hi(loop)`



## Input Structure: `directive`

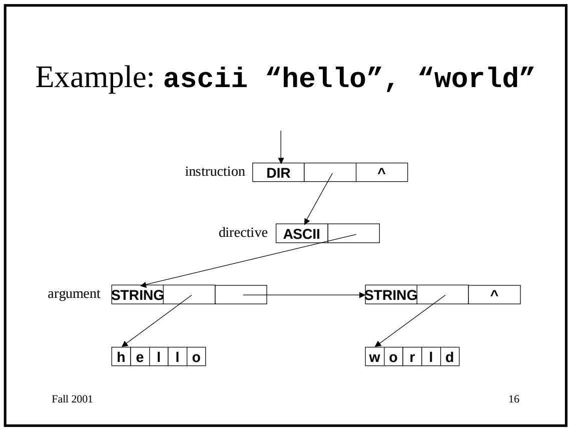
```
struct directive {
   Directive_Type dir_type; ——► ASCII, BYTE, …
   struct argument *arg_list;
};
```



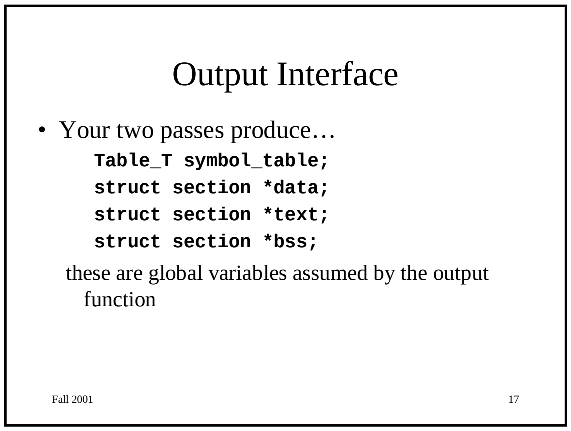
## Input Structure: `argument`

- Two types of arguments
  - string (e.g., **`“hello”`**)
  - expression (e.g., **`1+2`**)

```
struct argument {
   int arg_type; ——————————► STRING, EXP
   union {
      char *string;
      struct expression *exp;
   } u;
   struct argument *next;
};
```



## Example: **`ascii “hello”, “world”`**

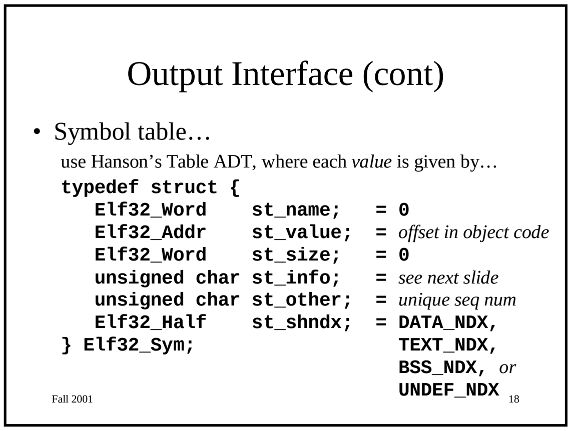


## Output Interface

- Your two passes produce…

```
Table_T symbol_table;
struct section *data;
struct section *text;
struct section *bss;
```

these are global variables assumed by the output function



## Output Interface (cont)

- Symbol table…

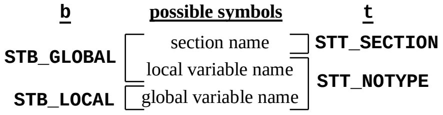
use Hanson’s Table ADT, where each *value* is given by…

```
typedef struct {
   Elf32_Word    st_name;   = 0
   Elf32_Addr    st_value;  = offset in object code
   Elf32_Word    st_size;   = 0
   unsigned char st_info;   = see next slide
   unsigned char st_other;  = unique seq num
   Elf32_Half    st_shndx;  = DATA_NDX,
} Elf32_Sym;                  TEXT_NDX,
                              BSS_NDX, or
                              UNDEF_NDX
```

## Output Interface (cont)

- To set **st_info** field of **Elf32_Sym** structure

  use **ELF32_ST_INFO(b,t)** macro, where<br>
  **b** specifies the symbol’s binding attribute<br>
  **t** specifies the symbol’s type



**b** possible symbols **t**

STB_GLOBAL

STB_LOCAL

section name

local variable name

global variable name

STT_SECTION

STT_NOTYPE

## Output Interface (cont)

- Each section…

```
struct section {
   unsigned int      obj_size;
   unsigned char     *obj_code;
   struct relocation *rel_list;
};
```

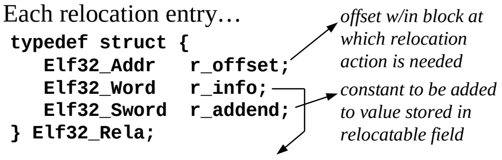
**obj_size** is given in bytes

**size = 0** and **obj_code = NULL** for BSS

```
struct relocation {
   Elf32_Rela rela;
   struct relocation *next;
}
```

## Output Interface (cont)

- Each relocation entry…



```
typedef struct {
   Elf32_Addr   r_offset;
   Elf32_Word   r_info;
   Elf32_Sword  r_addend;
} Elf32_Rela;
```

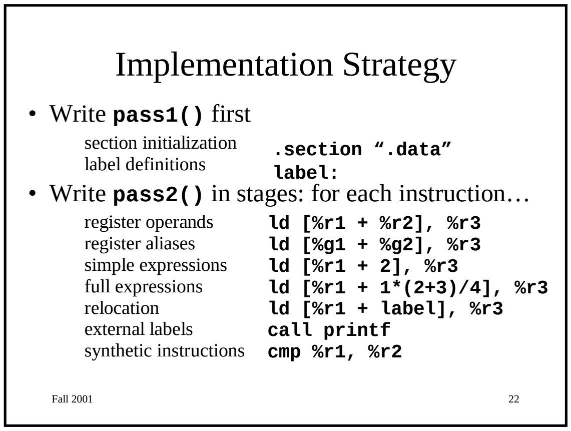
*offset w/in block at which relocation action is needed*

*constant to be added to value stored in relocatable field*

use **ELF32_R_INFO(s,t)** macro to set, where

**s** is the unique number for the symbol that is to be used (must match that entry’s **st_other** field)

**t** identifies the type of relocation action to be applied **R_SPARC_WDISP30, R_SPARC_WDISP22, R_SPARC_HI22,** or **R_SPARC_LO10**



## Implementation Strategy

- Write **`pass1()`** first

| | |
|---|---|
| section initialization | **`.section ".data"`** |
| label definitions | **`label:`** |

- Write **`pass2()`** in stages: for each instruction…

| | |
|---|---|
| register operands | **`ld [%r1 + %r2], %r3`** |
| register aliases | **`ld [%g1 + %g2], %r3`** |
| simple expressions | **`ld [%r1 + 2], %r3`** |
| full expressions | **`ld [%r1 + 1*(2+3)/4], %r3`** |
| relocation | **`ld [%r1 + label], %r3`** |
| external labels | **`call printf`** |
| synthetic instructions | **`cmp %r1, %r2`** |

Fall 2001 22