Some Examples: A9P3

Approach

- Rather than write out all the MIPS assembly language code, here I'll use pseudocode like

  `push($3)` to mean push the value stored in $3 onto the top of the system stack

  `$5 = pop()` to mean pop the top value of the stack and store the value in register $5

Some Examples: A9P4

Aproach

- for rule: *statements* $\rightarrow$ *PRINTLN LPAREN expr RPAREN*

Output

- **`println`** prints whatever is in \$1 on the screen, followed by a newline
- it overwrites (i.e. destroys) the contents of \$1 and \$31
- it is a library interface with the OS provided by the compiler
- print.merl has to be linked in, e.g.

```
./wlp4gen < source.wlp4i > source.asm
java cs241.linkasm < source.asm > source.merl
linker source.merl print.merl > exec.mips
```

- the directive **`.import print`** must be added to the prolog

Some Examples: A9P4

Aproach

- for rule: *statements→ PRINTLN LPAREN expr RPAREN*

Output

```
;;;code( println (expr); ) =
   ;; Prolog
   ;; $31 pushed on stack and print label imported
   ;; Body
   code(expr)           ; evaluate expr: $3 <- expr
   add $1, $3, $0       ; copy to $1: $1 <- expr
   lis $10              ; $10 <- print addr
   .word print          ;
   jalr $10             ; call print subroutine
   ;; Epilog
   ;; $31 restored
```

Some Examples: A9P5

Rules for Assignment

- dcls $\rightarrow$ dcls dcl BECOMES NUM SEMI
- dcl $\rightarrow$type ID
- e.g. int total = 0;

Notes

- code(NUM)
  - put the number, NUM, into register $3, i.e. **`$3` $\leftarrow$ `NUM`**
- code (dcl BECOMES NUM SEMI)
  - load NUM into $3
  - look up the offset of ID in the symbol table (i.e. the offset relative to the frame pointers $29)
  - generate the code: **`sw $3, ID_offset($29)`**

Some Examples: A9P5

Rules for Assignment

- statement → lvalue BECOMES expr SEMI
- lvalue → ID
- e.g. total = a+1;

Notes

- code(statement)
  - evaluate the expression expr by calling code(expr)
  - the results should be stored in register $3
  - look up the offset of the ID in the symbol table (i.e. the offset relative to the frame pointers $29)

```
code(statement) = code(expr)
                  sw $3, ID_offset($29)
```

Some Examples: A9P6

Rules for Comparison Test

- test $\rightarrow$ $expr_1$ LT $expr_2$

Notes

- there are two control structures in WLP4:
  - (1) while loops and (2) if-then-else statements
- both rely on comparison tests

Conventions

- \$0 $\leftarrow$ 0, no choice here, it's hardwired into MIPS
- \$11 $\leftarrow$ 1, we must add this to the prolog
- recall: when evaluating multiple expressions, in a recursively friendly way
  - results are returned in \$3
  - use stack (to store) and \$5 (to retrieve) intermediate results

Some Examples: A9P6

Rules for Comparison Test

- test $\rightarrow$ $expr_1$ LT $expr_2$

Generating Code

- evaluate the 1st expression, $expr_1$ (the results will be in \$3) and then push \$3 on the stack

```
code (expr1)      ; result is $3 ← expr1
push ($3)         ; top of stack ← expr1
```

- evaluate the 2nd expression, $expr_2$ (the results will be in \$3)

```
code (expr2)      ; result is $3 ← expr2
```

- pop off the stack results into \$5 and complete the test

```
$5 ← pop()        ; $5 ← expr1
slt $3, $5, $3    ; set $3 if expr1 < expr2
```

Some Examples: A9P7

Rules for Comparison Test

- test $\rightarrow$ $expr_1$ GT $expr_2$

Generating Code

- *note:* ($3 > $5) is the same as ($5 < $3)
- so by swapping the order of the source registers, e.g.

```
slt $3, $3, $5    ; $3 < $5
```

vs

```
slt $3, $5, $3    ; $3 > $5
```

- we can obtain the other comparison using one instruction
- So the code for test $\rightarrow$ $expr_1$ GT $expr_2$
  - is very similar to the code for test $\rightarrow$ $expr_1$ LT $expr_2$
  - except the order of the source registers are swapped

Some Examples: A9P7

Rules for Comparison Test

- test $\rightarrow$ $expr_1$ GE $expr_2$
- test $\rightarrow$ $expr_1$ LE $expr_2$

Generating Code

- *note:* (\$3 ≥ \$5) is the same as **not** (\$3 < \$5)
- *note:* (\$3 ≤ \$5) is the same as **not** (\$3 > \$5)
- Since the result of a **`slt`** comparison is either 0 or 1
- to take the *not* of the result, subtract it from 1 (i.e. \$11)

```
sub $3, $11, $3    ; $3 ← not($3)
```

- Why?
  - if \$3==1 (true), then 1-\$3==0 (false)
  - if \$3==0 (false), then 1-\$3==1 (true)
  - by CS241 convention, we will always store 1 in \$11

Some Examples: A9P7

Rules for Comparison Tests

- test $\rightarrow$ $expr_1$ NE $expr_2$

Code Generation

;; code(test) =

code($expr_1$)	; \$3 $\leftarrow$ $expr_1$
push(\$3)	; stack $\leftarrow$ $expr_1$
code($expr_2$)	; \$3 $\leftarrow$ $expr_2$
\$5 $\leftarrow$ pop()	; \$5 $\leftarrow$ $expr_1$
slt \$6, \$3, \$5	; \$6 $\leftarrow$ $expr_2 < expr_1$
slt \$7, \$5, \$3	; \$7 $\leftarrow$ $expr_1 < expr_2$
add \$3, \$6, \$7	; \$6 and \$7 cannot both be 1

- if $expr_1 == expr_2$, then both `slt` commands will return 0 and sum is 0. If one of the `slt` tests returns 1, the sum will be 1.

Some Examples: A9P7

Rules for Comparison Tests and the NOT operation

- test $\rightarrow$ $expr_1$ EQ $expr_2$

Code Generation

- do the code for $expr_1$!= $expr_2$ followed by the statement **`sub $3, $11, $3`**
- recall \$11 contains 1 and \$3 contains our results (a 0 or 1)
- again, subtraction (in this case) is equivalent to the NOT operation on the value in \$3.
  - it will flip a 0 to a 1 and a 1 to a 0, i.e.
    - if \$3 == 0 then \$11 - \$3 == 1
    - if \$3 == 1 then \$11 - \$3 == 0

Some Examples: A9P6 and P8

Automatically Generating Labels

- for control structures such as *while* loops and *if-then-else* statements you will need to be able to *generate unique labels*
- *idea:* have a function like label()
  - recall that the leading character must be a letter
  - each time it gets called, a variable gets incremented
  - its value is concatenated to a letter
    - e.g. L1, L2, L3, …
- to make the MIPS assembly language you generate easier to understand, could have separate labels to start and end
  - *while* loops: sw1, ew1, sw2, ew2, sw3, ew3, …
  - *then* branches of if-then-else: st1, et1, st2, et2, st3, et3, …

Some Examples: A9P6

Rules for While Loops

- statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE

Notes

- you will have to create a series of unique labels: st_wl1, st_wl2, st_wl3 , etc.

Code

```
code(statement ) = st_wl1:
                      code(test)
                      beq $3, $0, end_wl1
                      code(statements)
                      beq $0, $0, st_wl1
                   end_wl1:
```

Some Examples: A9P6

Rules for While Loops

- statement → WHILE LPAREN test RPAREN LBRACE statements RBRACE

Notes

- limited to jumping 2^{15} instructions forward
- limits the number of instructions created by the code(statements) line
- otherwise must do something like the following to jump farther

```
lis $6
.word end_wloop
jr $6
```

Some Examples: A9P8

Rules for If Statements

- statement $\rightarrow$ IF LPAREN test RPAREN LBRACE statements$_1$ RBRACE ELSE LBRACE statements$_2$ RBRACE

Notes

- you will have to create a series of unique labels: st_else1, st_else2, st_wl3 , etc.

Code

```
code(statement) =  code(test)                 ; $3 ← test
                   beq $3, $0, st_else1       ; if false do stmts2
                   code(statements1)
                   beq $0, $0, end_else1:     ; skip stmts2
                 st_else1:
                   code(statements2)
                 end_else1:
```

Some Examples: A9P8

Rules for If Statements

- statement $\rightarrow$ IF LPAREN test RPAREN LBRACE statements_1 RBRACE ELSE LBRACE statements_2 RBRACE

Code

```
code(statement) = code(test)                  ; $3 ← test
                  bne $3, $0, st_then1         ; if true do stmts1
                  code(statements2)
                  beq $0, $0, end_then1        ; skip stmts1
                st_then1:
                  code(statements1)
                end_then1:
```

- can put the “then” branch before or after the “else” statements branch.

Summary

Notes

- you now have all the ideas to generate code for Assignment 9!
- You can handle a single function that always takes two parameters and returns an integer.
- inside the body of the function you can have
  - additional *declarations* and *assignments* (e.g. a=1;)
  - *control structure*s {if-then-else , while loops}
    - using a variety of *comparison tests*: { <, <=, >, >=, ==, !=}
  - various *arithmetic operations* {+, -, *, /, %}
- *Hint:* generate comments with your code to aid debugging
- you are missing
  - pointers / memory allocation and deallocation
  - multiple procedures (i.e. one procedure calling another)