**LEC 10.5 Code Generation – Pointers & Procedures**

* **Pointers**
  + In order to support pointers, these must be supported:
    - NULL
      * Could use 0 – but dereferencing NULL should result in crash
      * So use 1 as NULL
    - Dereferencing operator
    - Address-of operator
    - Alloc/dealloc (i.e. new/delete)
    - Pointer arithmetic
    - Pointer comparison
    - Assignment through pointers
  + Dereferencing
    - i.e. \* expr
    - MIPS:

code(expr) ; $3 ← expr

lw $3, 0($3)

* + Comparison
    - Same for comparing ints
    - Use slt if expr1 & expr2 have type int
    - Use sltu if they have type int\*
    - Add a “type” field in each tree node to record the type
  + Pointer arithmetic
    - i.e. expr + term
    - MIPS:

; If expr, term : int

As before

; If expr : int\*, term : int

Code(expr)

Push $3

Code(term)

mult $3, $4

mflo $3 ; $3 = 4 \* term

pop $5 ; $5 = expr

add $3, $5, $3 ; $3 = expr + 4 \* term

* + - i.e. expr – term
    - MIPS:

; If expr, term : int

As before

; If expr, term : int\*

Code(expr)

Push $3

Code(term)

Pop $5

sub $3, $5, $3

div $3, $4

mflo $3 ; $3 = (expr – term) / 4

* + Assign thru pointer dereference
    - i.e. \* expr1 = expr2;
    - LHS = address at which to store the value
    - RHS = the value
    - Calculate the value of expr1
    - Use it as the address at which to store expr2
    - MIPS:

Code(expr2)

Push $3

Code(expr1)

Pop $5

sw $5, 0($3)

* + Address-of
    - i.e. & lvalue
    - MIPS:

; if lvalue = ID

lis $3

.word (look up ID’s offset in symbol table)

add $3, $29, $3

; if lvalue = \* expr

code(expr)

* + New/delete
    - Part of the runtime environment
    - Use the provided allocation module alloc.merl
    - Alloc.merl must be linked last
    - Add to the prologue:
    - .import init
      * Set up allocator’s data structures
      * Called once at the beginning of the asm file
      * Takes a param in $2
        + If calling program with mips.array, i.e. first param of wain is int\*
        + $2 = length of array
        + Otherwise $2 = 0
    - .import new
      * $1 = # of words needed
      * Returns a pointer to memory in $3
      * Returns 0 if allocation fails
      * i.e. new int[expr]
      * MIPS:

Code(expr)

add $1, $3, $0

call new

bne $3, $0, $11 ; skip next line if result of new != 0

add $3, $11, $0 ; if new failed, $3 = 1 (NULL)

* + - .import delete
      * $1 = address to be freed
      * Deleting NULL is guaranteed to be safe (does nothing)
      * First check the address; only call delete if address is not NULL
      * i.e. delete [] expr
      * MIPS:

Code(expr)

beq $3, $11, skipY ; skip if $3 = 1 (NULL)

add $1, $3, $0

call delete

skipY:

* **Procedures**
  + Make sure to execute main first
    - Either generate the code for main at the top, followed by procedures
    - Or generate a first line that jumps to main, i.e. beq $0, $0, wain
  + Main prologue/epilogue
    - Save $1, $2 on stack
    - Import print, init, new, delete
    - Set $4, $11 etc. constant registers
    - Save & set $29
    - Call init
    - Restore stack at the end & jr $31
  + Procedure prologues/epilogues
    - Don’t need to import
    - Don’t need to set constants
    - Save & set $29
    - Restore stack at the end & jr $31
  + Procedures should save & restore all registers that are modified
  + How to know which registers to save if the procedure code hasn’t been generated yet?
    - Don’t save $3 – used for output
    - Can just save all the registers
    - Or since our code generation methods don’t use any registers past $8, just save these
  + Saving frame pointer
    - Let caller save $29 (and restore it after returning from call)
    - Let callee set $29 (to its own stack frame)
  + Make sure to name functions differently than compiler-generated labels
    - Prefix all function labels with F, e.g. “Ff”, “Fg”, “Ffunction”
  + Parameters
    - Could use registers, but might not be enough
    - Could also use the stack
  + Code for procedure call
  + i.e. ID ( expr1 , expr2 , … exprn )
  + MIPS:

push $29

push $31

code(expr1)

push $3

code(expr2)

push $3

…

code(exprn)

push $3

lis $5

.word FID

jalr $5

pop all arguments

pop $31

pop $29

* + Consider:

int g(int a, int b, int c) {

int d = 0; int e = 0; int f = 0;

…

}

* + g's symbol table: (variable → offset from $29)
    - a → 0 // args
    - b → -4
    - c → -8
    - d → -12 // local vars
    - e → -16
    - f → -20
  + Params should have positive offsets from $29
  + Local vars should have negative offsets from $29
  + i.e. need to fix symbol table before code generation
  + Add 4 \* the # of regs to each offset in table
  + Also save registers after pushing local vars
  + g's new symbol table:
    - a → 12 // args
    - b → 8
    - c → 4
    - d → 0 // local vars
    - e → -4
    - f → -8
  + OR let caller save registers
    - But code would be written once per call
  + Code for procedure definition
  + i.e. int ID ( params ) { dcls statements return expr ; }
  + MIPS:

sub $29, $30, $4 ; set $29

push dcls

push registers

code(statements)

code(expr) ; $3 ← expr

pop registers

add $30, $29, $4 ; restore stack pointer

jr $31