CS 241

Sibelius Peng

Contents

1 May 7

Sequential Programs: nothing fancy, no parallel, concurrency, multi-threading
Start point: bare hardware
for 241, simulated MIPS machine. Only interprets 0's and 1's.
... at end: get `C`-like programs to run on MIPS

Binary & Hexadecimal

bit: a single 0 or 1
byte: 8 bits $2^8 = 256$ different patterns
nibble: 4 bits
word - 241 architecture: 32-bits
common place in the real world: 64-bits

1010 what does this mean?

- 10 - unsigned binary
- -2 - "sign-magnitude" binary
- -6 - 2's complement
- newline - ASCII
- gray - grayscale (0000 black, 1111 white)

  ⋮

The meaning is in the eye of the beholder (which eye?)

Files

- header
- file extensions

Programming: type declarations - interpret the bits a certain way.
can you change how bits are interpreted? - casting - be careful

Decimal (base 10) $12349 = 1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 9 \times 10^0$ (digits 0...9)

Binary (base 2) (digits 0..1) $11001001 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^3 + 1 \times 2^0 = 201_{10}$
201 convert to binary:

$$\begin{array}{rr} 201 - 128 = 73 & 1 \\ 73 - 64 = 9 & 1 \\ 9 - 32 & 0 \\ 9 - 16 & 0 \\ 9 - 8 = 1 & 1 \\ \vdots & \end{array}$$

How do we represent negative numbers?

- use first bit: 1 negative, 0 positive $\implies$ "sign-magnitude" binary
- addition and subtraction are difficult
- two zeros: positive and negative $\implies$ wasteful

$11001001 = -(64 + 8 + 1) = -73$

1.1 2's Complement

1. Interpret the n-bit number as an unsigned integer
2. If the first bit is zero, done
3. Else subtract 2^n

eg $n = 3, 2^n = 8$

000	001	010	011	100	101	110	111
0	1	2	3	$4 - 8 = -4$	$5 - 8 = -3$	-2	-1

So n bits represent -2^{n-1} to $2^{n-1} - 1$

- only 1 zero
- left bit gives sign
- addition is clean - just arithmetic mod 2^n

Alternative:

- positive numbers are simply binary magnitude
- negative
    1. start with magnitude of number
    2. flip bits: $1 \to 0, 0 \to 1$
    3. add 1

eg -73 to 8-bit binary
magnitude: 01001001 (73 in binary)
flip bits: 10110110
add 1: 10110111 (2's complement representation of -73)

eg What does 11001001 represent in 2's complement?

- reverse process?
  - subtract 1
  - flip bits
  - ⋮
- do process again

Exercise: show these two are equivalent

soln 11001001 - negative
flip bits: 00110110
add 1: 0010111 (magnitude) $\implies$ 55
result: -55

Given a byte 11001001, how do we tell if this is unsigned binary (201), sign-magnitude (-73), 2's complement (-55)? We don't.

A character ASCII (we will use), others

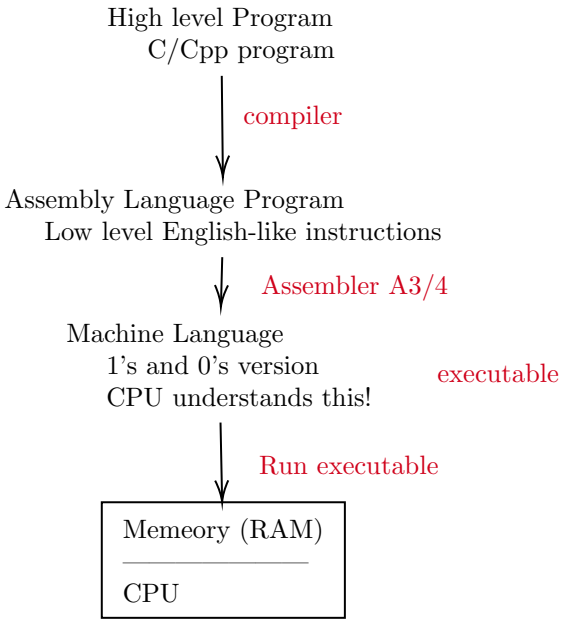
An Instruction certain 23-bit patterns represent MIPS machine code instructions

Garbage ...

2

May 9

2.1 Overview



3 Jun 18

Ex

$$E \to E + T | T$$
$$T \to T * F | F$$
$$F \to a | b | c | (E)$$

- left associative
- precedence (ops)

Is this $LL(1)$? No
TOS[1] and next input symbol - is there a choice of rule?
Let's say E to a

$$E \implies T \implies F \implies a$$
$$E \implies E + T \implies T + T \implies F + T \implies a + T$$

Why? Left recursion $E \to E + T \quad E \to T$.
Two derivations, same first symbol.
$*$ Left recursion, always not $LL(1)$

$$E \to T + E | T$$
$$T \to F * T | F$$
$$F \to a | b | c | (E)$$

right recursive
TOS: E, input: a..., still not $LL(1)$

Need to factor

$$E \to TE'$$
$$E' \to \epsilon | + E$$
$$T \to FT'$$
$$T' \to \epsilon | * T$$
$$F \to a | b | c$$

This is $LL(1)$ but is at odds with left recursion. Left assoc.
See next parsing alg.

[1] top of stack

Ambiguous

$$S \to a|b|c|SOS|(S)$$
$$O \to +|-|*|/$$

$E \to E\ O\ T|T$
$T \to a|b|c|(E)$
$O \to +|-|*|/$
unambiguous, left associative

$E \to EAT|T$
$T \to TMF|F$
$F \to a|b|c|(E)$

$A \to +|-$
$M \to *|/$
operator precedence

3.1 Bottom-Up Parsing

- go from w to S.

Stack store partially reduced information read so far.

$$w \Longleftarrow \alpha_k \Longleftarrow \alpha_{k-1} \Longleftarrow \ldots \Longleftarrow S$$

Invariant: stack + entire unread input = α_i (or w or S)

Choices at each step:

1. Shift character from input onto stack
2. Reduce TOS in the RHS of a grammar rule: replace with LHS

Accept if stack contains only S' when input is ϵ. Equivalent: $\vdash S \dashv$ on empty input - accept when machine pushing $\dashv$

How do we know whether to shift or reduce?
Use next char of input to help. Problem is still hard.

Theorem (Donald Knuth, 1965)
The set $\{wa|\exists x, S \implies {}^*wax\}$
w is stack, a is next input char is Regular Language.
$\implies$ can be described by a DFA.

Use a DFA to make shift/Reduce Decisions
Results in LR parsing

- left-to-right through input
- Rightmost Derivation

Defn An item is a production with a dot, $\bullet$, somewhere on the RHS (indicate partially completed rules)

$$\vdots$$

missed sth due to cineplex interview....

4

Jun 20

Notes

- label transitions with the symbol follows the dot advance the dot in the next state.
- If the dot precedes a non-terminal A, add all productions with A on the LHS to the state (dot in leftmost position)

4.1 Using Automaton

continued from slide 10
Backtracking in the DFA - must remember the path we followed - also push states onto the stack as well

LR(0) * if not in a reduce state, simply shift • follow transition for that symbol. If no transition $\implies$ ERROR. Reduce: only 1 rule

If any item $A\alpha\bullet$ occurs in a state in which it is not alone, then there is a shift-reduce or reduce-reduce conflict and the grammar is not LR(0)

5 June 27

5.1 Build Symbol Table

- Traverse the parse tree to collect variable declarations
  - for each node corresponding to the rule: `dcl -> TYPE ID`
  - extract ID's name and type (int, int*) and add it to the symbol table
  - if name already exists in table $\implies$ error
  - multiple declarations checked
- Traverse parse tree
  - check for `factor -> ID` and `lvalue -> ID`
  - if ID's name is not in the symbol table, ERROR
  - undeclared variables checked

  You must do these all in one pass

5.2 Implementing Symbol Table

- map
- global variable

```
map <string, string> symbolTable; // name -> type
```

BUT • doesn't account for scope, • ot procedures

Issues

```
int f() {
   int x = 0;
   int y = 0;
   return x;
}
int wain(int a, int b) {
```

```
   int x = 0; // okay
   // return y; // not ok
   return x; // ok
}
```

Permit duplicate declarations in different procedures
Forbid duplicate declarations in same procedures
Also

```
int f() {...} // overloading
int f() {...} // not okay in wlp4
```

$\Longrightarrow$ need a separate symbol table for each procedure
Have a "top-level" symbol table that stores all procedure names

```
map <string, map<string, string> > topSymbolTable;
   //procedure name, symbol table
```

When traversing the parse tree

- Find node corresponding to rule:
  `procedure -> INT ID LPAREN ...`
  `main -> INT WAIN ...`

$\Longrightarrow$ new procedure

  - make sure its name not already in symbol table
  - if not, create new entry

  Implementation: may want a global variable to store "current procedure"

  - update each time find `procedure ->` or `main ->`

  For variables, store: declared type & name in Symbol Table

Do procedures have a type? Yes - signatures
• return type for WLP4 is only INT • parameter types
$\Longrightarrow$ So signature is only param list types
Store this in the top-level symbol table

```
map<string /*proc name*/, pair<vector<string> /*signature*/, map<string, string>/*
    symbol table*/ > > > topSymbolTable;
```

Tp compute the signature:

- `paramlist -> dcl`
- `paramlist -> dcl` COMMA paramlist
- (if `param ->`, then signature is empty)

All of this analysis, can be done in a single pass (traverse of tree)

Types Why do programming languages have types?

Recall: from only bits - don't know what they represent. Type tells us how to interpret the bits.

A good type system prevent us from re-interpreting the bits as sth else

```
// ex
int *p = NULL;
p = 7; // ERROR type mismatch
```

casting

WLP4, 2 types, int, int*

To check type correctness, need to

- determine the type associated with each variable/expression
- ensure that all operators are applied to operands of the correct type.

Ex `d = a + (b + c);`

name	type
a	int*
b	int
c	int
d	int*

How do we determine type? Declarations

- `dcl -> TYPE ID`
- add a field in the symbol table

  ...

Catching type ERRORS

- determine the type of every expression by applying type rules given by language spec
- if no rule applicable, or if an expression type does not match its context $\implies$ ERROR

```
string typeof(Tree &t) {
   for each c : t.children
      compute typeof(c)
   use t.rule to decide what type rule is relevant
      combine types of children
      determine the type of t
      if not possible: ERROR
}

string typeof(Tree &t) {
   if t.rule == "ID name"
      return symboltable.lookup(name);
      ...
}
```

6 July 4

Loperand	op	Roperand	Resulting type
int	+	int	int
int*	+	int	int*
int	+	int*	int*

More on here

Procedures

- body must be well-typed
- must return int

wain 1st dcl can be int or int*, 2nd dcl must be int, body must be well-typed, return type must be int

Lvalues

- LHS and RHS of an assignment statement $x = y$ are treated differently
- RHS denotes a value
- LHS denotes storage destination.
  must name a memory location.

Expressions that denotes storage locations are lvalues. eg $x, y.$ etc

- variable names
- dereferenced pointers
- any lvalue surrounded by ( )

these formats are enforced by the WLP4 grammar (syntax)

6.1 Code Generation

```
         parse tree        parse tree/symbol table
parsing  ----------> semantic -------------->      code gen ----> assembly
                     analysis
```

How many (equivalent) Assembly programs are there for a given WLP4 program? infinite
Properties of code generated

- correctness
- easy
- efficient - compiler runtime - program runtime (how fast it runs)
- for 241 optimization: fastest = fewest instructions

Ex Input:

```
int wain(int a, int b){return a;}
```

conventions

- parameters of wain are held in $1 and $2
  loaders are mipstwoints and mipsarray
- output will be passed in $3

```
add $3, $1, $0
jr $31
```

Symbol table

Name	Type	Location
a	int	$1
b	int	$2

should add field to sym tab where each symbol is stored
Where should local variables/parameters be stored?

- choice: registers (faster, not many registers, may run out) or Memory (RAM) (stack) general scheme is to store all of these on stack. including $1 and $2 from wain

7 July 9

for simplicity, store all local var/param on stack

- including params of wain
- symbol table store name, type, offsets

∗ you are not evaluating/executing the input code, you are only translating it into equivalent Assembly.

```
int wain(int a, int b) {return a;}
```

```
sw $1, -4($30)
sw $2, -8($30)
lis $4
.word 4
sub $30, $30, $4
sub $30, $30, $4
lw $3, 4 ($30) ; lookup in symbol table ; return a
add $30, $30, $4
add $30, $30, $4
jr $31
```

symbol table

name	type	offset
a	int	4
b	int	0

7.1 Problem

```
int wain(int a, int b) { int c=0; return a;}
```

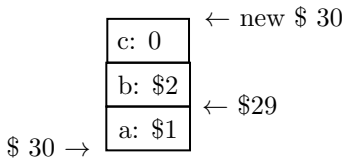
can't know offsets until all declarations are processed, because $30 changes with each new declaration.

Two new conventions

- $4 always contains 4
- $29 points to the bottom of the stack frame.
- If offsets are calculated w.r.t $29, then they will be constant.

```
lis $4
.word 4
sub $29, $30, $4
sw $1, -4($30) ; push a
sub $30, $30, $4
sw $2, -4($30) // push b
sub $30, $30, $4
sw $0, -4($30)//  space for c=0 on stack
sub $30, $30, $4
lw $3, 0 ($29)
add $30, $30, $4
add $30, $30, $4
add $30, $30, $4
jr $31
```

name	offset (from $29)
a	0
b	-4
c	-8



7.2 More complicated

```
int wain(int a, int b) {return a + b;}
```

In general, for each grammar rule $A \to \alpha$. build code for A, code(A) from code(α)

Convention

- use $3 for "output" of all expressions

ex a+b:
$3 ← eval(a)
$3 ← eval(b)
$3 ← $3 + $3

Need a place to store pending computations.

- use a register?

```
code(a)
add $5, $3, $0
code(b)
add $3, $5, $3
```

need 1 extra reg for temp values

What about `a+(b+c)`

```
code(a) // $3 <- a
add $5, $3, $0 // $5 <- $3
code(b) // $3 <- b
add $6, $3, $0 //$6 <- $3
code(c) // $3 <- c
add $3, $6, $3 // $3 <- b + c
add $3, $5, $3 // $3 <- a + (b + c)
```

need 2 extra reg for temp values

ex $a + (b + (c + d))$ How many extra regs? 3
May run out of registers $\implies$ use the stack instead. • general solution!

```
code(a)
push($3)
code(b)
push($3)
code(c)
push($3)
code(d)
pop($5)
add $3, $5, $3 // $3 <- c+d
pop($5)
add $3, $5, $3 // $3 <- b+ (c+d)
pop($5)
add $3, $5, $3 // $3 <- a + (b+ (c+d))
```

only need 1 extra
In general: $expr_1 \to expr_2 + term$

```
code(expr_1) =
   code(expr_2)
   + push($3)
   + code(term)
   + pop($5)
   + "add␣$3,␣$5,␣$3"
```

singleton rules usually easy
$S \rightarrow$ BOF procedures EOF
code(S) = code(procedures)
expr $\rightarrow$ term
code(expr) = code(term)

7.3 Print

`println(expr);` Prints value of expr and a newline

Implementation: A2 p6, 7a
Runtime environment: set of procedures supplied by compiler (or OS) to assist programs in their execution: e.g. msvcrt.dll libc.so

Make print part of Runtime Env - you need to link it in

```
wlp4gen < source.wlp4i > source.asm
cs241.linkasm < source.asm > source.merl
linker source.merl print.merl > source.mips
mips {twoints, array} source.mips
```

Notes

- $1 is input to print
  - if $1 holds sth else - save $1 and restore later
  - calling print: clobbers $31 - save and restore $31

`code(println(expr));`

8 july 11

```
code(println(expr)); =  code (expr)
                       add $1, $3, $0
                       sw $31, -4 ($30)
                       sub $30, $30, $4
                       lis $5
                       .word print
                       jalr $5
                       add $30, $30, $4
                       lw $31, -4 ($30)
                       lw $1, 0 ($29) // if desired
```

Assignment statement (stmt)

- statememt → expr1 BECOMES expr2 SEMI
  For now, only int $\implies$ only ID

```
code(stmt) = code(expr2) // $3 <- expr2
             sw $3, ___ ($29) // lookup offset in symbol table
```

- if and while - need boolean testing
- suggested convention
  - store 1 in $11
  - also store print in $10

Code so far

```
.import print // prologue
lis $4
.word 4
lis $11
.word 1
lis $10
```

```
.word print
sub $29, $30, $4

// (allocate space on stack for all vars)

// YOUR CODE
add $30, $29, $4 ; Epilogue
jr $31
```

Boolean tests $test \rightarrow expr_1 < expr_2$

```
code(test) = code(expr_1) // $3 <- expr_1
             add $5, $3, $0 // $5 <- $3
             code(expr_2) // $3 <- expr_2
             slt $3, $5, $3
```

$test \rightarrow expr_1 > expr_2$ implement expr2 ¡ expr1
$test \rightarrow expr_1 \neq expr_2$

```
code (test) =  code(expr1)
             add $5, $3, $0
             code(expr2)
             slt $6, $3, $5 // at most one of these
             slt $7, $5, $3 // comparisons is true, not both
             add $3, $6, $7
```

$test \rightarrow expr_1 == expr_2$ treat as NOT (expr1 != expr2)
add `sub $3, $31, $3` to above $3 ← 1 - $3

8.1 If statements

$$stmt \rightarrow IF\ test\ stms_1\ ELSE\ stms_2$$

```
code (stmt) = code(test)
         + beq $3, $0, else
         + code (stms1)
         + beq $0, $0, endif
   + else:  + code (stms2)
   + endif:
```

Issue

- need to generate unique label names
- keep counter X for it stms
- use labels: else X, endif X, true X
- increment X for each new if stmt

Alternative

```
code (stmt) = code(test)
      + bne $3, $0, true
      + code (stms2)
      + beq $0, $0, endif
   + true:  + code (stms1)
   + endif:
```

8.2 While

$$stmt \rightarrow WHILE\ (test)\ \{stms\}$$

use counter Y to generate fresh labels

```
code(stmt) = loopY:
         code(test)
         beq, $3, $0, doneY
         code(stms)
         beq $0, $0, loopY
      doneY:
```

9 Pointers

Need to support

- NULL
- deference
- address of
- comparisons
- pointer arith
- alloc/dealloc
- assignments through pointers

5 to go

```
int *p = NULL;
if(p) // false
if(*p) // crash
```

NULL

- could use 0
- go to 0x0 and get value does not crash
- should use a number that is not divisible by 4, say 1

$factor \rightarrow NULL$ `code(factor) = add $3, $11, $0`

deref $factor \rightarrow *expr$ - valid address

```
code(factor) = code(expr)
               lw $3, 0 ($3)
```

Comparisons same as int comparisons

- no negative pointers $\implies$ use sltu instead of slt
- How do we know type of expr?
  - rerun `typeof` in A8 to check if int* or int
  - Better: save type in a field for each node of tree

Pointer arithmetic $expr_1 \rightarrow expr_2 + term$
meaning is dependent on types involved

Type

	expr2 +	term		
1.	int	int	$\implies$	as before
2.	int*	int	$\implies$	expr2 + (4 * term)
3.	int	int*	$\implies$	(4*expr2)+term

for 2

```
code (expr1) = code(expr2)
             push($3)
             code(term)
             mult $3, $4
             mflo $3
             pop($5)
             add $3, $5, $3
```

	expr2 -	term		
1.	int	int	$\implies$	as before
2.	int*	int	$\implies$	expr2 - (4 * term)
3.	int	int*	not valid	
4.	int*	int*	$\implies$	(expr2-term)/4

9.1 Assignment through pointer deref

LHS = address at which we store the value
RHS = the value

$stmt \rightarrow \underbrace{ID}_{lvalue}\ BECOMES\ expr2\ SEMI$
$stmt \rightarrow \underbrace{STAR\ expr1}_{lvalue}\ BECOMES\ expr2\ SEMI$

- calc value of expr1
- use as address to store value of expr2

```
code(stmt) = code(expr2)
           push($3)
           code(expr1) // without *
```

```
            pop($5)
            sw $5, 0 ($3)
```

Address-of: 2 cases: ID, STAR expr
factor → AMP

&a if expr = ID

```
code(factor) =
   lis $3
   .word __ // look up in the symbol table
   add $3, $29, $3
```

&*a if expr = STAR expr2

```
code(factor) = code (expr2)
```

Delete - part of runtime environment
we procide allocation module : alloc.merl

- link same as print
- link last

Add to prologue

```
.import init
.import new
.import delete
```

Function init sets up the initial data structure

- Must be called exactly once at the beginning of your Assignment file
  - call init in the prologue
  - takes parameter in $2
  - if calling with `mips.array` : $2 = length of array
    else $2 = 0

9.2 new and delete

new

- $1 = number of words requested
- return ptr to memory in $3
- returns 0 if alloc not possible

```
code(new int[expr]) =
   code(expr)
   add $1, $3, $0
   call(new)
   bne $3, $0, 1  // if result is 0, set to NULL (1)
   add $3, $11, $0
```

delete $1 = ptr to be dealloc

```
code(delete[] expr) =
   code(expr)
   beq $3, $11, skipDelete // if NULL do nothing
   add $1, $3, $0
   call(delete)
skipDelete:
```

small note: the reason why NULL = 0x1, here

10 Compiling Procedures

10.1 Big picture

```
int f() {...}
int g() {...}
int wain( , ) { ... }
```

⇓

```
// prologue for main (wain)
// main function (wain)
// epilogue for main



f: ... // prologue-specific prologue/epilogue
jr $31

g: ...
jr $31
```

Main Prologue/epilogue

- save $1, $2 on stack
- import print, int, new, delete
- set $4, $11, etc
- set $29
- call init // $2 <- 0 ?

- reset stack to bottom
- jr $31

Procedure-specific prologues

- don't need imports, set constants, etc
- set $29
- save registers that the proc will overwrite

- restore regs, reset stack to end
- jr $31

10.2 Saving and restore regs

- Procedures should save and restore all regs that it modifies
- How do we know which registers to save?
  - if not sure, save & restore all of them! except $3
  - Our code gen uses: `$1 - 7, $10, $11, $29-31`
    if your code gen uses others, - okay, but need to keep track of regs used
  - don't forget to save reg $29

10.2.1 Two approaches to saving registers

caller-save vs callee-save

Suppose f calls g

- caller-save: f saves all the registers containing critical data, then calls g
- callee-save: g saves all registers that it modifies

Our approach has been:

- caller-save for $31
- callee-save for everything else

Different approaches also work

Q who saves $29? caller or callee?

Suppose callee, g, saves $29

```
g: sub $29, $30, $4
   saves g's regs
```

2 tasks in g's prologue: point $29 to g's frame; save regs
Which one do we do first

1. save regs and then set $29
   $29 for will be based on $30 and # regs saved
2. set $29 first, then save regs
   $30 hasn't changed yet
   easy to set $29 to $30 - 4, then save regs
   easy to implement

Q How do you save $29?

need to save old $29 (f) before we overwrite/update to new $29 (g)?

so

```
g:
   push($29)
   add $29, $30, $0
   push other regs
```

OR let caller, f, save $29 before procedure call

```
f:
   push($29)
   push($31)
   call(g)
   pop($31)
   pop($29)
```

next issue: labels - what if my WLP4 prog is:

```
int init() {...}
int print() {...}
```

- procedure names match the names in the runtime environment
- duplicate labels
- won't compile

More generally, what if a function has the same name as one of labels we generate?

Fix make sure if never happens

- use a naming scheme that prevents duplication
- for functions f, g, h, etc. use the labels Ff, Fg, Fh, etc.
  i.e. reserve labels starting with F as denoting user defined functions
  Then make sure your compiler does not generate other labels starting with F

10.3 Parameters

registers (may run out.) or stack? stack.

Registers fast, don't have lw, sw, limited #

stack lots of space, this is what we will do
factor -> ID(expr_1, ..., expr_n)
f calls g

```
code(factor) =
   push($29) // save bottom of stack frame for f
   push($31)
   code(expr_1)
   push($3)
   ...
   ...
   ...
   code(expr_n) // allocates space on stack
   push($3)     //   for params with initial values args
   lis $3                // call g
   .word Ffunction_name  // call g
   jalr $5
   pop all args
   pop($31)
   pop($29)
```

procedure -> INT ID (params) {dcls stms RETURN expr;}

```
code(procedure) =
   sub $29, $30, $4 // set bottom of stack frame
   push regs
   code(dcls)
   code(stms)
   code(expr)
   pop regs
   add $30 $29, $4
   jr $31
```

Listing 10.1: first idea captionpos

Problem

- g's params below $29
- g's local variables above $29
- ∗ save regs between params and local vars

Fix

- saved regs between
- swap order: do dcls first, then save regs

fix offset

- offset below $29: +ve
- offset above $29: -ve

add 4 × # args to all offset in symbol table. See Fig. 10.1

Figure 10.1: f calls g

Alternative

Suppose we had:
each call to g saves and restores the registers it will modify - callee-save

```
f() {
   ... // save f's reg
   g();
   g(); // save regs once per "call site"
   g();
   g();
   ... // restore here
}
```

Listing 10.2: caller-save

Does this save on number lines generated in code gen?

11 Optimization

- very large problem-complicated
- in general; minimize runtime
- in cs241: # lines code

$\Longrightarrow$ computationally unsolvable: but we can use heristics

Ex `code 1+2`

```
lis $3
.word 1
sw $3, -4($30)
sub $30, $30, $4
lis $3
.word 2
lw $5, 0($30)
add $30, $30, $4
add $3, $5, $3
```

Listing 11.1: 9 words

```
lis $3
.word 3
```

- have the compiler:
  recognize 1,2 are constants $\Longrightarrow$ is also constant
- instead of gen code to compute at runtime compiler can do the evaluation at compile time

called constant folding

11.1 constant propagation

```
int x = 2;
return x + x;
```

```
lis $3           // ---
.word 2          // int x = 2;   Do I need this?
sw $3, -4($20)   //              if this is the only place x is used, NO
sub $30, $30, $4 // ----
lw $3,__($29)    // here could recogenize that x = 2
push($3)         // 2 + 2 = 4
lw $3,__($29)    // lis $3
pop($3)          // .word 4
add $3, $5, $3
```

Listing 11.2: 11 words

11.2 common subexpression slimination

even if x's value is unknown, could recognize $3 already contains x

```
lw $3,__($29)
add $3, $3, $3
```

$(a+b)*(a+b)$

- use a reg to hold $a+b$.
- mult by itself instead of generating the code to evaluate $a+b$ twice

11.3 Dead code elimination

if you are certain that some branch of a program will never be reached, don't generate that code

11.4 Register Allocation

- cheaper to use regs instead of stack - save sw, lw

  ex $14 - $28 unused by our code gen
  - most used var
  - problem: &, address-of, if saved in a reg, what is &? - needs a RAM address

Problem Cont. & (address of) if a var is stored in a reg, it doesn't have a RAM address, so what does & return? If you need the address, you need the address should store in RAM

11.5 Strength Reduction

add usually runs faster than mult (in real world)
for cs241: mult by 2:

```
lis $5      //
.word 2     //
mult $3, $5 // => add $3, $3, $3
mflo $3     //
```

Real world: bit shift

11.6 Procedure-specific optimization

11.6.1 Inlining

```
int f(int x) {
   return x + x;
}

int wain(int a, int b) {
   return f(a);
}
```

↓

```
int wain(int a, int b) {
   return a + a;
}
```

- replace the function call with its body, right in caller
- saves overhead of calling the function

Do we need to generate code for function f? Ff: ...
Don't if it is inlined in all calls to f.

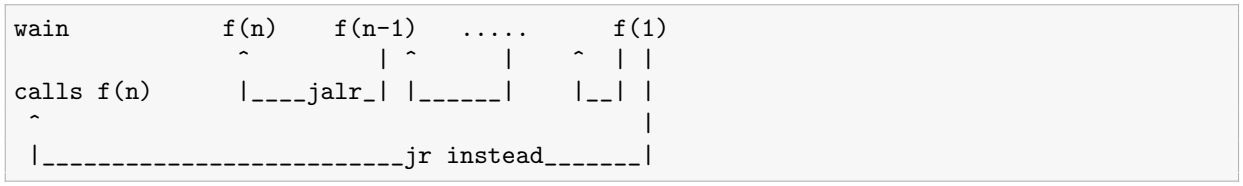
Downside if f is called many times, body of f is copied to many places

- inlining saves on overhead of calling functions
- if inlined in all places function is called $\implies$ do not need to generate code for f.

Compare with cost of copying the body in place of function calls.

11.6.2 Tail Recursion

```
int f(int n) {
   ...
   return f(n-1);
}
```



- can reuse the stack frame
- recursive calls have same number of params, local vars
- for successive calls use jr instead of jalr
- don't need to save $31

12 Memory Management & Heap

- If you want data to out live its scope ("persists")
  copy it to another scope, i.e. to a stack-allocated variable in an outer scope

- OR don't use the stack-use heap

```
----------
| code   |
----------
| heap   |
----------
|   ||   |
|   \/   |
|        |
|   /\   |
|   ||   |
----------
| stack  |
----------
```

Ex

```
C * f() {
   C *d = new C; // ob on heap
   return d;
}
```

- stack contains a pointer to heap memory
- heap objects live on after the stack frame, they have been allocated in, is popped
- To release heap data, must call free/delete

12.0.1 new/delete (malloc/free)

- variety of implementations
- List of free blocks • maintain a linked list of pointers to free areas of the heap
- Initially - entire heap is free, so linked list has 1 entry.

Suppose heap is 1k

$pic1$

Request 16 bytes

- actually allocate 20 bytes: 16 bytes + 1 int (4 bytes)
- return a pointer to 2nd word

12.1 Implicit Memory Management: Garbage Collection

Java, Racket - reclaim memory when it is no longer accessible

Data structures to manage new/delete

- linked list of free blocks
- there are other ds's

```
//eg1
int f() {
   MyClass ob = new MyClass();
   ...
} // ob is out of scope, no more references to heap object, it can be reclaimed

// eg2
int f() {
   if (x == y) {
      MyClass ob1 = new MyClass();
      ob2 = ob1;
   } // ob1 is out of scope, but ob2 store addr to block on heap
   ...
} // ob2 is out of scope, no more references to heap object, it can be reclaimed
```

12.1.1 GC techniques

1. Mark and Sweep
   - scan entire stack, look at pointers
   - for each pointer found, mark the heap block it pointing to
   - if heap block contains pointers, follow then as well, mark, etc.

   Then scan heap, reclaim any blocks not marked and clear all marks.
2. References Counting
   - for each heap block, keep track of the number of pointers that point to it
   - Must watch every ptr, and update ref count
     each time a pointer is reassigned: decrement old, increment new
   - If count reaches 0, reclaim it

Problem circular references: both have ref count 1 but are collectively inaccessible

3. Copying Garbage Collector
    - Heap is divided into two halves "from" and "to"
    - allocate only from "from"
    - When "from" fills up, all reachable data is copied from "from" to "to" and roles are reversed
    - Built in compaction - guaranteed that after each swap, all reachable data will occupy contiguous memory, so no fragmentation
    - Downside: heap is only half sized

memory management is not free

13 Linker & Loader

13.1 Loaders

- load (copy it into RAM) your program into RAM to start executing it
- may load program P into a memory address α
  where α may not be 0x0

$\Longrightarrow$ labels may be resolved to the wrong memory addresses
loader will need to fix it.

miss some

start of july 30
a pic

The output of most Assemblers is not pure machine code

- it's object code, MERL for cs241
- object file contains binary code + auxiliary info. needed by the loader (and later linker)
  relocation entries

mips.twoints/array

- optimal 2nd argument = address at which load mips file. Typically the relocation is done by the loader.

Still possible to write programs that only work if loaded to 0x0

```
top:
   lis $5
   .word top
   beq $0, $5, ...

// ---------------
lis $5
.word 12
jr $5
jr $31
```

If you want to relocatable code, always use labels to specify jump targets

```
lis $5
.word jump
jr $5
jump: jr $31
```

13.2 Linker

- convenient to store code in multiple files
- code should be relocatable $\implies$ MERL format

pic

- a linker needs to intelligently merge MERL files
- you should not expect programmers to use unique labels in different files

Merl - external symbol reference (ESR)

- format code 0x11
- location (address) in the code/MERL file
- name of symbol