HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 7:
Simple Code Generation

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Announcements

- New TF! Nicholas Hasselmo
- CS Nights: Mondays 8pm-10pm in MD 119. Pizza provided!

- Project 2 out
  - Due Thursday Oct 4 (9 days remaining)
- Project 3 released today
  - Due Tuesday Oct 9 (14 days)
- Project 4 will be released Tuesday Oct 2

Today

- Code generation: mapping F-ish code to MIPS code
  - Variables
  - Nested expressions
  - Statements
  - Improving things:
    - Simple constant folding
    - Expressions for conditional branches
    - Register allocation for binary expressions

Preliminaries

- Fortran programming language
  - Name from **Form**ula **Tran**slation
  - Originally developed by IBM in 1950s for scientific and engineering applications
  - One of first high-level programming languages
    - i.e., a replacement for hand-coding assembly
  - Influenced C programming language
  - Early version had no functions or procedures
  - Current versions still popular for high-performance computing
- Our source language is Fish (**F**ortran-**ish**)
  - No functions/procedures, imperative, structured control flow
- Our target language is MIPS assembly

Source

- Expressions

```
type exp =
      Var of var
    | Int of int
    | Binop of exp * binop * exp
    | Not of exp
    | Or of exp * exp
    | And of exp * exp
    | Assign of var * exp
```

Source

- Statements

```
type stmt =
      Seq of stmt * stmt
    | If of exp * stmt * stmt
    | While of exp * stmt
    | For of exp * exp * exp * stmt
    | Exp of exp
    | Return of exp
```

MIPS

```
type label = string

type reg =
   R0 | R1 | R2 | … | R31

type operand =
   Reg of reg
 | Immed of word
```

MIPS

```
type inst =
    Add of reg * reg * operand
  | Li of reg * word
  | Slt of reg * reg * operand
  | Beq of reg * reg * label
  | Bgez of reg * label
  | J of label
  | La of reg * label
  | Lw of reg * reg * word
  | Sw of reg * reg * word
  | Label of label | ...
```

Variables

- Fish has only global variables
- Initial approach: put each variable in the *data segment*
  - Part of object file that contains program's initialized data
  - Data segment is loaded into memory when object file loads
  - `.data` directive instructs assembler to put data in data segment
  - E.g.,

```
.data
.align 0
x:  .word 0
y:  .word 0
z:  .word 0
```

`.align n` means align next datum on 2^n byte boundary. `.align 0` turns off alignment

`x`, `y`, and `z` are labels of memory locations, each of which is initialized to 4-bytes of zero

Variable Access

- To compile `x = x + 1`

  (i.e., the Fish AST `Assign("x", BinOp(Var("x"), Plus, Int 1))`)

```
la $3, x       ; load x's address into reg $3
lw $2, 0($3)  ; load x's value into reg $2
addi $2,$2,1  ; add 1 to reg $2
sw $2, 0($3)  ; store value back in x
```

First Problem: Nested Expressions

- Consider
  `Binop(Binop(“x”,Plus,”y”),Plus,Binop(“w”,Plus,”z”))`
  - i.e., `(x + y) + (w + z)`
- Target language doesn’t have nested expressions, just 3-operand assembly instructions!
  - `add` *`rd, rs, st`*
- How do we compile nested expressions?

A Simple Strategy

- Given `Binop(`*`A`*`, Plus, `*`B`*`)`
  - Translate sub-expression *`A`* so that the result is stored in a register (e.g., `$3`)
  - Translate subexpression *`B`* so that the result is stored in a different register (e.g., `$2`)
  - Generate `add $2, $3, $2`

- Any problems?
- What if we have a deeply nested expression, with more subexpressions than we have registers?

A Slightly Less Simple Strategy

- Key idea: always put result in `$2`, and save result to memory
- Given `Binop(`*`A`*`, Plus, `*`B`*`)`
  - Translate sub-expression *`A`* so that the result is stored in `$2`
  - Save `$2` to memory
  - Translate subexpression *`B`* so that the result is stored in `$2`
  - Restore *`A`*'s result to, say, `$3`
  - Generate `add $2, $3, $2`

Example

- `Binop(Binop("x",Plus,"y"),Plus,Binop("w",Plus,"z"))`
- 1. Compute x+y, putting result in `$2`
- 2. Store `$2` into temporary `t1`
- 3. Compute w+z, putting result in `$2`
- 4. Load temporary `t1` into register, say `$3`
- 5. `add $2, $3, $2`

Expression Compilation

```
let rec exp2mips(e:exp):inst list =
    match e with
    | Int j -> [Li(R2, Word32.fromInt j)]
    | Var x -> [La(R2,x), Lw(R2,R2,zero)]
    | Binop(e1,b,e2) ->
      (let t = new_temp() in
         (exp2mips e1) @ [La(R3,t), Sw(R2,R3,zero)]
        @(exp2mips e2) @ [La(R3,t), Lw(R3,R3,zero)]
        @(match b with
            Plus -> [Add(R2,R2,Reg R3)]
          | ... -> ...))
    | Assign(x,e) ->  [exp2mips e] @
                      [La(R3,x), Sw(R2,R3,zero)]
```

Statement Compilation

```
let rec stmt2mips(s:stmt):inst list =
   match s with
   | Exp e ->
       exp2mips e
   | Seq(s1,s2) ->
     (stmt2mips s1) @ (stmt2mips s2)
   | ...
```

Statement Compilation

```
| If(e,s1,s2) ->
  (let else_l = new_label() in
   let end_l = new_label() in
   (exp2mips e) @ [Beq(R2,R0,else_l)] @
   (stmt2mips s1) @ [J end_l,Label else_l] @
   (stmt2mips s2) @ [Label end_l])
```

```
       E
       beq $2, $0, ELSE
       S1
       j     END
ELSE:  S2
END:   ...
```

Statement Compilation

```
| While(e,s) ->
    (let test_l = new_label() in
     let top_l = new_label() in
       [J test_l, Label top_l] @
       (stmt2mips s) @
       [Label test_l] @
       (exp2mips e) @
       [Bne(R2,R0,top_l)])
```

```
       j     TEST
TOP:   S
TEST:  E
       bne $2, $0, TOP
```

Statement Compilation

```
| For(e1,e2,e3,s) ->
    stmt2mips(Seq(Exp e1,While(e2,Seq(s,Exp e3))))
```

```
for (e1; e2; e3) { S }
```

is equivalent to

```
e1; while (e2) { S; e3; }
```

Inefficiencies

- We've got a translation from Fish to MIPS assembly!
- But the translation has lots of inefficiencies...
  - No constant folding
    - e.g., `Plus(Int 35, Int 7)` could be translated to `Int 42`
  - Inefficient use of expressions in control flow
    - e.g., `if (x == y)` *`S1`* `else` *`S2`* is translated by evaluating `x == y` and then doing a `beq` comparing it to 0. Could directly do a `beq` on `x` and `y`
    - e.g., `if (`*`E1`* `&&` *`E2`*`)` *`S1`* `else` *`S2`* could lazily evaluate *`E1`* `&&` *`E2`*: if *`E1`* is 0, jump directly to *`S2`* instead of computing *`E2`*
  - Lots of `la`/`lw` and `la`/`sw` to handle variables and temporaries
  - Always write subexpression's result to temporary, even if could keep it in a register

Constant Folding: Take 1

```
let rec exp2mips'(e:exp) : inst list =
 match e with
    Int w -> [Li(R2, Word32.fromInt w)]
  | Binop(e1,Plus,Int 0) -> exp2mips' e1
  | Binop(Int i1,Plus,Int i2) ->
      exp2mips' (Int (i1+i2))
  | Binop(Int i1,Minus,Int i2) ->
      exp2mips' (Int (i1-i2))
  | Binop(e1,b,e2) -> ...
```

- What's wrong with this?
- What about `7 + (42 - 42)`?
- How could we fix it?

Conditional Contexts

- Consider `if (x < y) then S1 else S2`
- Translates to

```
        [put x in $3, and y in $2]
        slt  $2, $3, $2
        beq  $2, $0, ELSE
        [instructions for S1]
        j END
    ELSE:
        [instructions for S2]
    END:
```

- In most contexts for an expression, we want a value
- But for conditionals, we use the comparison to jump to a label and don't otherwise use it
- May be able to avoid materializing value

Translate Expressions in Conditionals Specially

```
let rec bexp2mips(e:exp) (t:label) (f:label) =
  match e with
    Int 0 -> [J f]
  | Int _ -> [J t]
  | Binop(e1,Eq,e2) -> let tmp = new_temp() in
      (exp2mips e1) @
      [La(R3,tmp), Sw(R2,R3,R0)] @
      (exp2mips e2) @
      [La(R3,tmp), Lw(R3,R3,R0),
       Bne(R3,R2,f), J t]
  | ...
```

Global Variables

- We treated all variables (including temporary variables) as if they were global
  - Set aside space in data segment, with label
  - To read: load address of label, then load value stored at address
  - To write: load address of label, then store value to that address
- Inefficient!
  - E.g., `x+x` requires loading `x`'s address twice!
  - Lots of memory operations
- How could we do better?

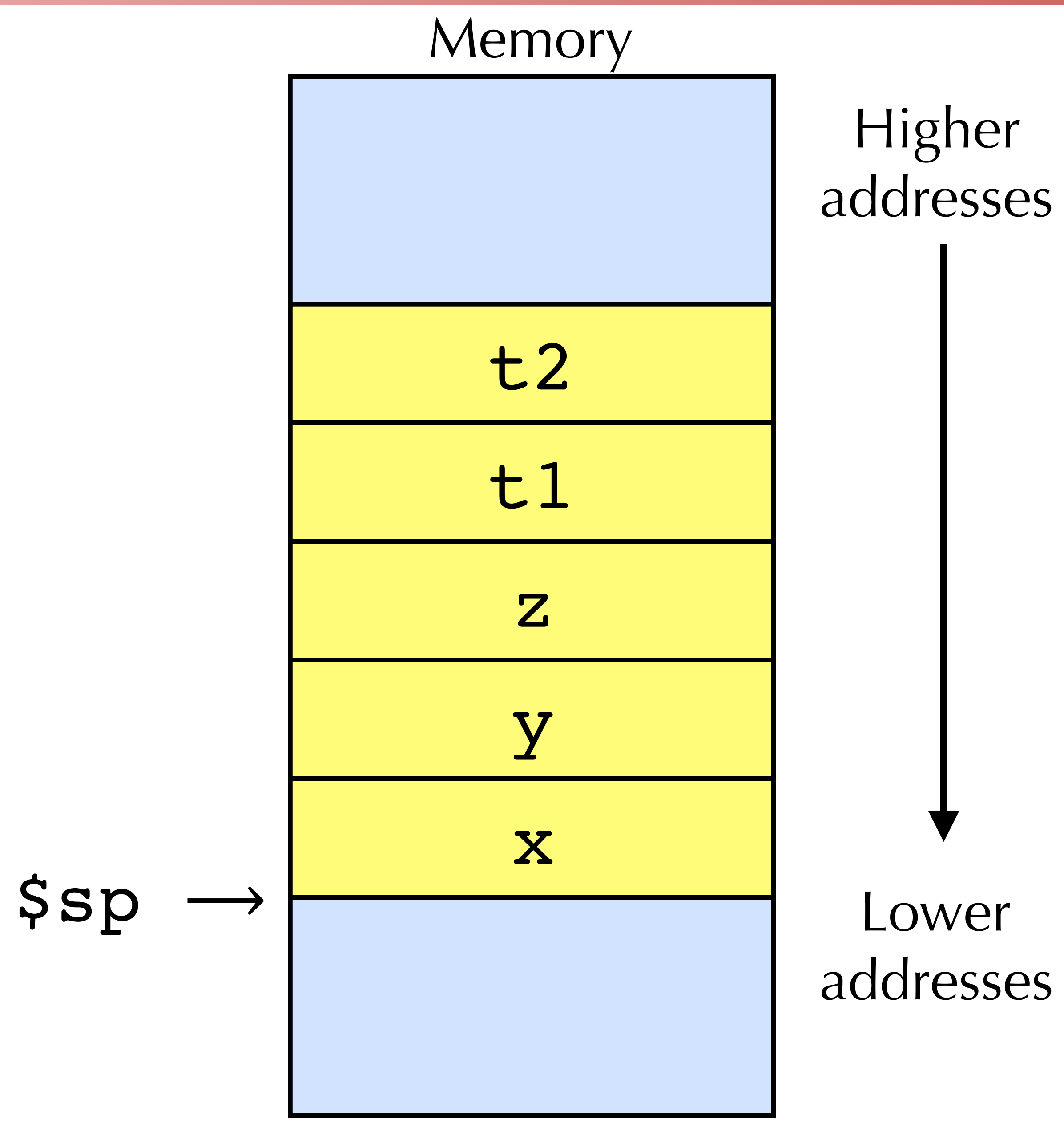
Register Allocation

- One option: use registers to hold variable's value
  - No need to access memory in order to use variable!
- But, what if more variables than registers?
  - Won't be able to avoid some memory accesses for variables
- But can we at least avoid loading addresses?

- (More later in course on register allocation!)

Frames

- Key idea:
  - Set aside one block of memory for all variables
  - Each variable corresponds to an offset within block
  - Set register `$29` (aka `$sp`, for **s**tack **p**ointer) to start of block
  - Access variable *v* at address `$sp` + [offset for *x*]



[x↦0, y↦4, z↦8, t1↦12, t2↦16]

Before and After

- Translating `z = x + 1`

Before

```
la   $3,x
lw   $2,0($3)
addi $2,$2,1
la   $3,z
sw   $2,0($3)
```

After

```
lw   $2,0($sp)
addi $2,$2,1
sw   $2,8($sp)
```

Lowering

- Get rid of nested expressions before translating
  - Introduce new variables to hold intermediate results
  - Perhaps do things like constant folding
- For example, `a = (x + y) + (z + w)` might be translated to

```
t0 := x + y;
t1 := z + w;
a := t0 + t1;
```

12 instructions (9 memory)

```
t0 := x + y;        lw   $v0, <xoff>($sp)
                    lw   $v1, <yoff>($sp)
                    add  $v0, $v0, $v1
                    sw   $v0, <t0off>($sp)
```

```
t1 := z + w;        lw   $v0, <zoff>($sp)
                    lw   $v1, <woff>($sp)
                    add  $v0, $v0, $v1
                    sw   $v0, <t1off>($sp)
```

```
a := t0 + t1;       lw   $v0, <t0off>($sp)
                    lw   $v1, <t1off>($sp)
                    add  $v0, $v0, $v1
                    sw   $v0, <aoff>($sp)
```

Still inefficient

- Doing lots of loads and stores
- We should not need to load/store from temps!
  - (Or from variables, but we'll deal with those later)
- Another idea: Use registers instead of temp variables to hold intermediate values
- But of course we have only finite registers, and expressions could be deeply nested
- So use just, say, k registers to hold first k temps

Example

```
t0 := x;           # load variable
t1 := y;           # load variable
t2 := t0 + t1;     # add
t3 := z;           # load variable
t4 := w;           # load variable
t5 := t3 + t4;     # add
t6 := t2 + t5;     # add
a  := t6;          # store result
```

Example

```
t0 := x;             lw   $t0,<xoff>($sp)
t1 := y;             lw   $t1,<yoff>($sp)
t2 := t0 + t1;       add  $t2,$t0,$t1
t3 := z;             lw   $t3,<zoff>($sp)
t4 := w;             lw   $t4,<woff>($sp)
t5 := t3 + t4;       add  $t5,$t3,$t4
t6 := t2 + t5;       add  $t6,$t2,$t5
a  := t6;            sw   $t6,<aoff>($sp)
```

- Note that each little statement can be directly translated to MIPS instructions
- 8 instructions, 5 of them memory!

Re-using Temps

```
t0  := x;          # t0 in use
t1  := y;          # t0,t1 in use
t2  := t0 + t1;    # t2 in use       t0,t1 freed
t3  := z;          # t2,t3 in use
t4  := w;          # t2,t3,t4 in use
t5  := t3 + t4;    # t2,t5 in use t3,t4 freed
t6  := t2 + t5;    # t6 in use       t2,t5 freed
a   := t6;         #                    t6 freed
```

- We could reuse temps that are no longer in use!

Re-using Temps

```
t0  := x;            # t0 in use
t1  := y;            # t0,t1 in use
t0  := t0 + t1;      # t0 in use          t1 freed
t1  := z;            # t0,t1 in use
t2  := w;            # t0,t1,t2 in use
t1  := t1 + t2;      # t0,t1 in use       t2 freed
t0  := t0 + t1;      # t0 in use          t1 freed
a   := t0;           #                    t0 freed
```

- Variables in use behave like a stack...
- Why?

More Re-use of Temps

- Consider `a=(x+y)*x`

```
t0 := x;
t1 := y;
t0 := t0 + t1;
t1 := x;
t0 := t0 * t1;
a  := t0;
```

Requires a memory load (`t0 := x;`)

Requires another memory load for same value! (`t1 := x;`)

- How could you avoid the redundant memory load?

More Re-use of Temps

- Consider `a=(x+y)*x`

```
t0 := x;
t1 := y;
t1 := t0 + t1;

t0 := t1 * t0;
a  := t0;
```

← No need to reload `x`, it is still in `t0`

Register Allocation

- We will study register allocation in more detail later in course
- But key ideas for now:
  - For each temp, calculate **live range**
    - Variable t is live at a program point if, on control flow path, there is subsequent read of t without an intervening write
    - (In functional code, variables are never re-defined, making it simpler)
  - Calculate which variables are live at the same time
    - These variables can't be allocated to same register

Register Allocation ctd

- Key ideas, ctd:
  - ...
  - Draw **interference graph**: nodes are variables, edge between variables if they are live at same time
  - Color graph: each color is a register; nodes that are live at same time can't have same color/register
  - Graph coloring is register allocation!
- What if more variables than registers? i.e., graph coloring not possible?
  - There's the rub...