CS 536 Announcements for Wednesday, April 17, 2024

Last Time

- continue code generation
  - function declaration, call, and return
  - expressions
  - literals
  - assignment
  - I/O

Today

- wrap up code generation
  - tuple access
  - control-flow constructs
- introduce control flow graphs

Next Time

- optimization

P6 : Codegen class

Constants for registers and logical constants

e.g., `FP , SP , T0 , T1` RA V0 A0 Codegen.FP → "$fp"

Methods to help automatically generate code

`generate(opcode, ... args ... )`

e.g., `generate("add", "$t0", "$t0", "$t1")`

writes out `add $t0, $t0, $t1`

versions for fewer args as well

`generateIndexed(opcode,arg1, arg2, offset)`

e.g., `generateIndexed("lw", "$t0", $t1", -12)`

writes out `lw $t0, -12($t1)`

`genPush(reg) / genPop(reg)` → 2 MIPS instrs each

`nextLabel()` – returns a unique string to use as a label → of the form .Lx

↑ unique int

`genLabel(L)` – places a label generates the code

↑ string

.L3:

if ".L3" is contents of L

Code Generation for Tuple Access

Offset from base of tuple to certain field is known statically

- compiler can do the math for the slot address
- not true for languages with pointers!

Example

```
tuple Inner {
    logical hi.
    integer there.
    integer c.
}.

tuple Demo {
    tuple Inner b.
    integer val.
}.

void f{} [
    tuple Demo inst.

    ... = inst:b:c.

  inst:b:c = ... .
```

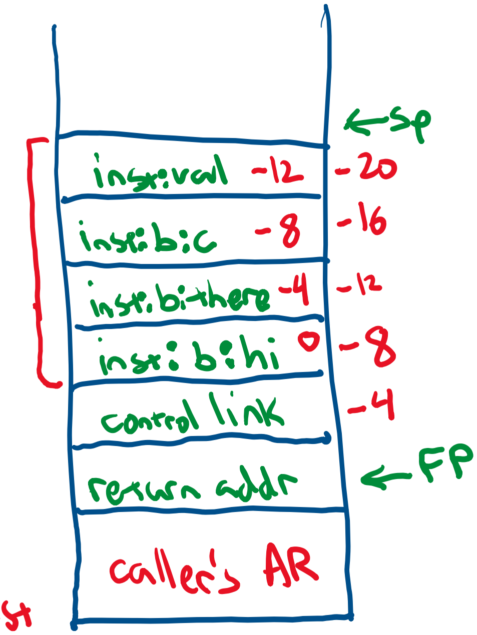
12 bytes

16 bytes

space for inst

inst is based at $fp-8

field for b:c is -8 off the base of inst



RHS - put value on stack

```
lw $t0, -16($fp)  # $t0 = value stored at $fp-16
push $t0
```

LHS - put address on stack

```
subu $t0, $fp, 16  # $t0 = $fp-16
push $t0
```

Control flow graphs

Kinds of control flow

- function calls – saw last lecture (jal, jr)
- selection – if, if-else, if-elseif, switch
- repetition – while, do-while, repeat until, for
- short-circuited operators &, |

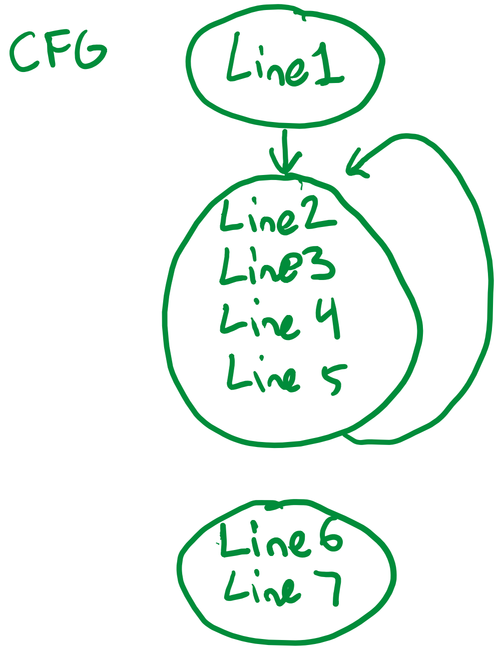
Control flow graph (CFG)

- important representation for program optimization
- helpful way to visualize source code

Example

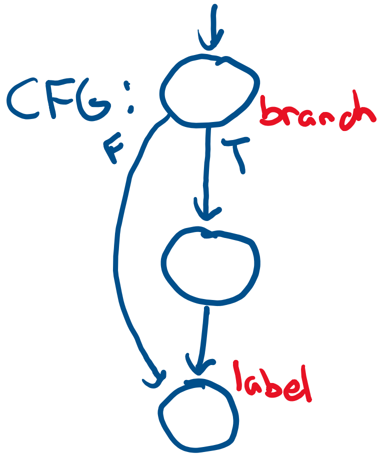
```
Line1: li $t0, 4
Line2: li $t1, 3
Line3: add $t0, $t0, $t1
Line4: sw $t0, val
Line5: b Line2
Line6: sw $t0, 0($sp)
Line7: subu $sp, $sp, 4
```

```
        t0 = 4
Line2:  t1 = 3
        t0 = t0 + t1
        val = t0
        goto Line2
        push t0 on stack
```

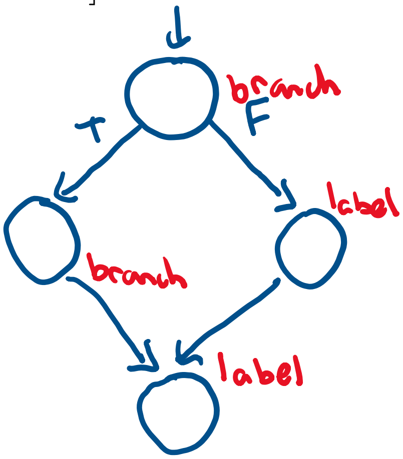


Kinds of control flow in base

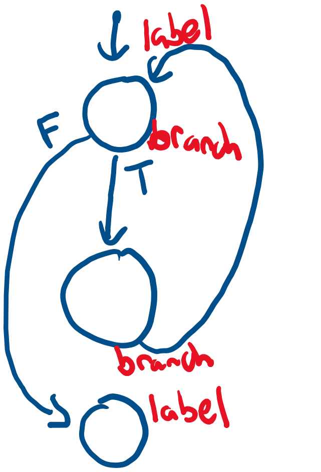
```
if exp [
   …
]
```



```
if exp [
   …
] else [
   …
]
```



```
while exp [
   …
]
```



What is needed at the assembly-code level

- branching
  - unconditional
  - conditional
- labels

MIPS

```
b label
beq r1, src, label
```

use branch in if/while control structures (rather than jump)

register or immediate value

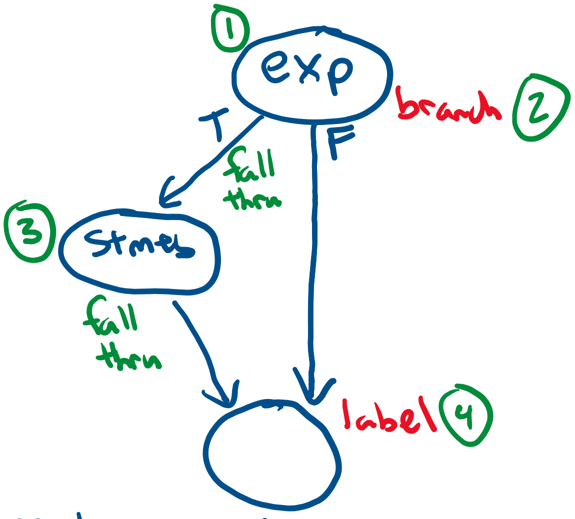
Also: bne, bgt, bge, blt, ble

Code generation for `if` statements

base code example:

```
    if a == b [
        $ body of if
    ]
```

Need to linearize - output a sequence of instructions



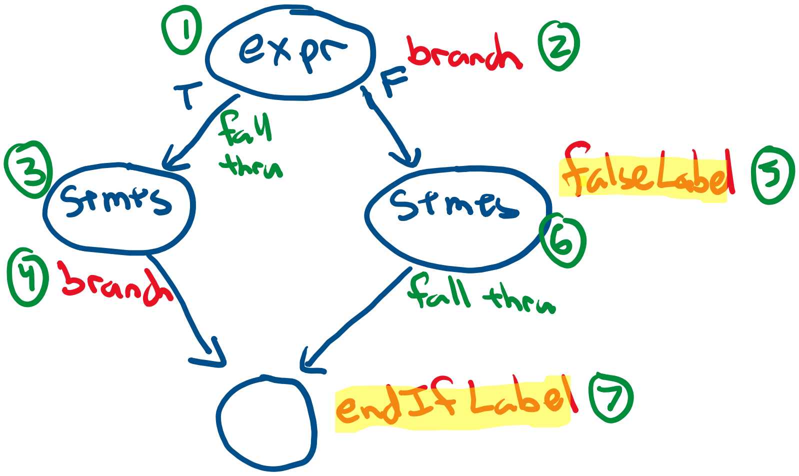
Code generation steps:

- get a label for end of construct
- 1 generate code for expression -leaving result on stack
- 2 generate conditional branch - to label (not yet placed!)
- 3 generate body of `if`
- 4 place end-of-construct label

Code generation for `if-else` statements

base code example:

```
    if a > b [
        $ body of if
    ]
    else [
        $ body of else
    ]
```



Need these labels to be unique, ie, generated by Codegen.nextLabel()

Code generation for `if-else` statements (cont.)

base code:

```
if a > b [
    $ body of if
]
else [
    $ body of else
]
```

MIPS code outline:

```
     lw $t0, addr_a       codeGen
     push $t0             on Id(a)

     lw $t0, addr_b       codeGen
①    push $t0             on Id(b)

     pop $t1
     pop $t0                  codeGen
     sgt $t0, $t0, $t1        on >
     push $t0

②    pop $t0
     beq $t0, FALSE, falseLabel
                   constant in Codegen.java
     .
③    .  # body of if
     .
④    b doneIfLabel

⑤ falseLabel:
     .
⑥    .  # body of else
     .

⑦ doneIfLabel:
```

Use Codegen.java
- has push/pop methods to generate MIPS code

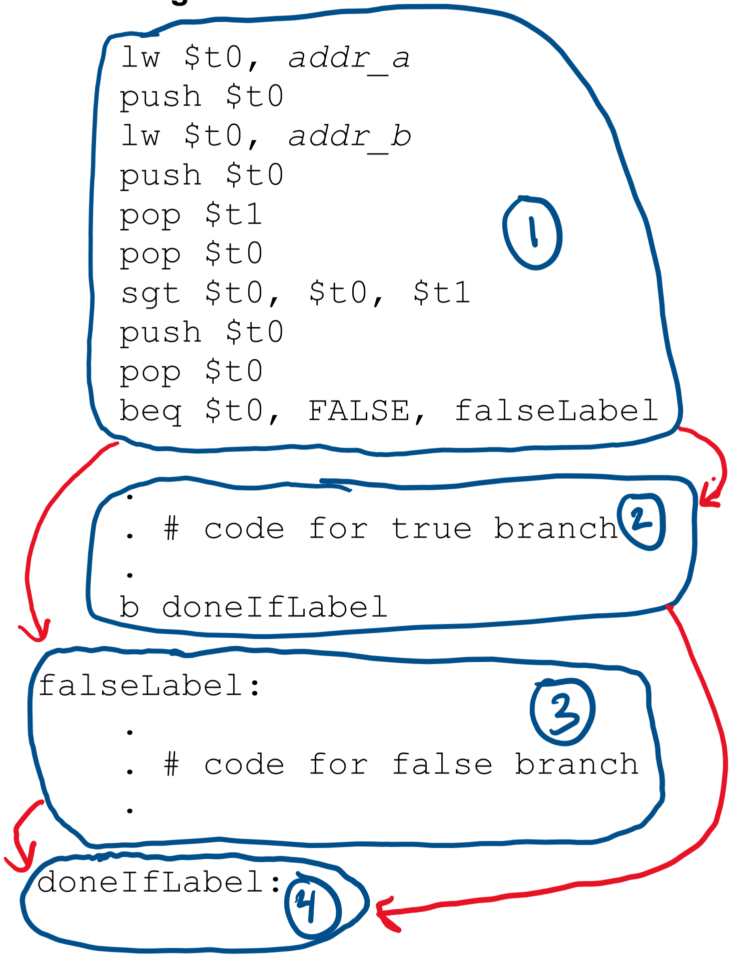
sgt R2, R0, R1
sets R2 to 1 if R0>R1
to 0 otherwise

Also have: sge, slt, sle, seq, sne

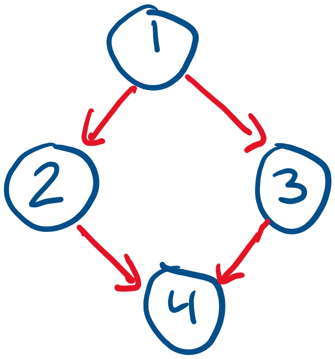
Note: only ended up using beq & b branching instrs

Code generation for `if-else` statements (cont.)

Revisiting the CFG



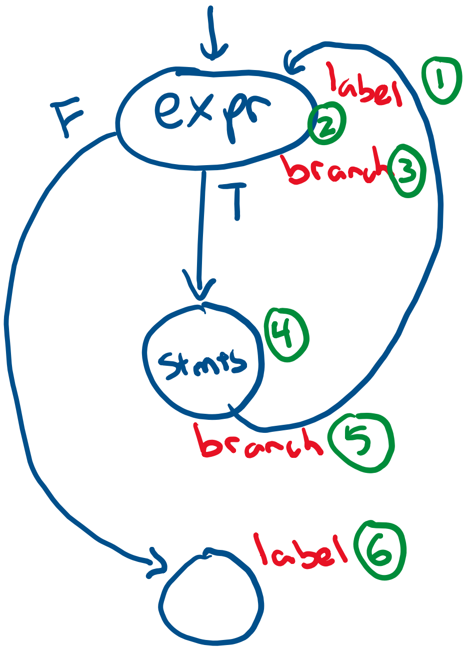
```
lw $t0, addr_a
push $t0
lw $t0, addr_b
push $t0
pop $t1
pop $t0
sgt $t0, $t0, $t1
push $t0
pop $t0
beq $t0, FALSE, falseLabel
.
. # code for true branch
.
b doneIfLabel
falseLabel:
    .
    . # code for false branch
    .
doneIfLabel:
```



Code generation for `while` statements

base code example:

```
    while a == b [
        $ body of while
    ]
```



MIPS tips

It’s really easy to get confused with assembly

Some suggestions

- start simple: main procedure that prints the value 1
    - get procedure `main` to compile and run
        - function prologue and epilogue
    - trivial case of expressions: evaluating the constant 1, which pushes a 1 on the stack
    - printing: `write << 1.`
- then grow your compiler incrementally
    - 1 expressions
    - 2 control constructs
    - 3 call/return

Create super simple test cases

- main procedure: print the value of some expression
- create more and more complicated expressions

Regression suite

- rerun **all** test cases to check whether you introduced a bug
- more suggestions
    - try writing desired assembly code by hand before having the compiler generate it
    - draw pictures of program flow
    - have your compiler put in detailed comments in the assembly code it emits