Code Generation for Control Flow Constructs

Code generation for control flow

1. MIPS for Function call
Reserve return value slot.
Put arguments on the stack;
Jump to callee's label.
Retrieve result value.

2. Control flow graph (CFG)
Basic block: node in the CFG. Largest run of instructions that will always be executed in sequence.

3. MIPS for If-then stmt: One jump.
   MIPS for if-then-else stmt: Two jumps.
   MIPS for while loop: Two jumps.
   MIPS for do-while loop: One jump.
   Technique: use negated condition to save jumps.

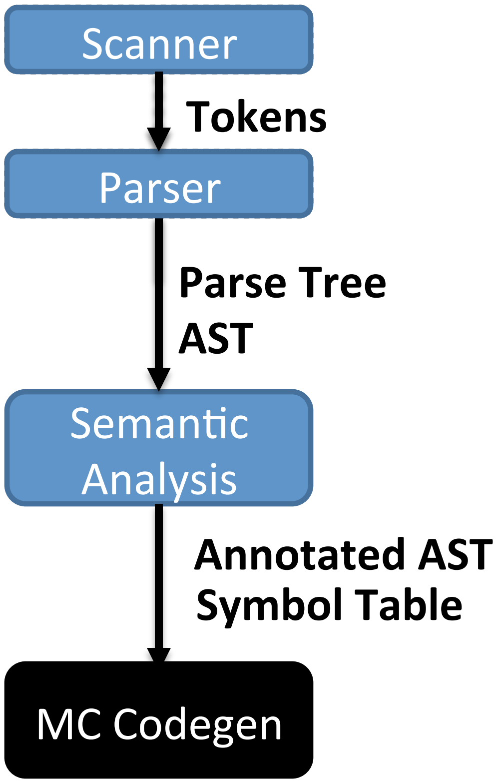
Roadmap

Last time:

- Got the basics of MIPS
- CodeGen for *some* AST node types

This time:

- Do the rest of the AST nodes
- Introduce control flow graphs



A Quick Warm-Up: MIPS for `id = 1 + 2;`

push 1	`li $t0 1`
	`sw $t0 0($sp)`
	`subu $sp $sp 4`

push 2	`li $t0 2`
	`sw $t0 0($sp)`
	`subu $sp $sp 4`

pop opR	`lw $t1 4($sp)` `addu $sp $sp 4`
pop opL	`lw $t0 4($sp)` `addu $sp $sp 4`
Do 1+2	`add $t0 $t0 $t1`
push RHS	`sw $t0 0($sp)` `subu $sp $sp 4`

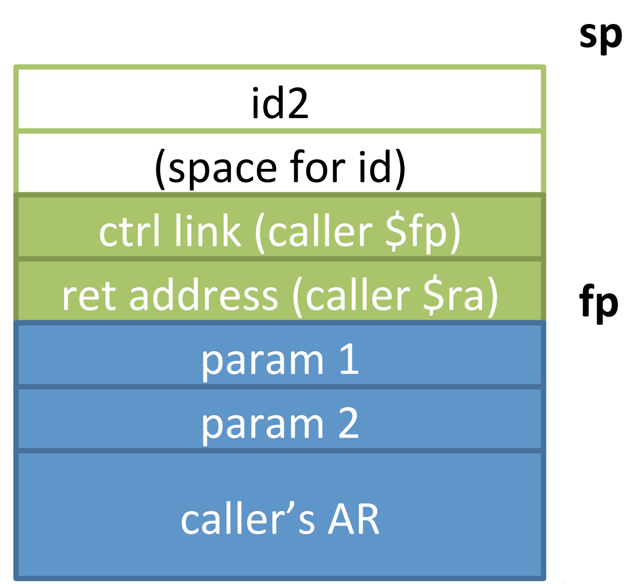
push LHS	`la $t0 -8($fp)`
	`sw $t0 0($sp)`
	`subu $sp $sp 4`

pop LHS	`lw $t1 4($sp)` `addu $sp $sp 4`
pop RHS	`lw $t0 4($sp)` `addu $sp $sp 4`
Do assign	`sw $t0 0($t1)`

This is an assignment, so the LHS should be the location of the variable. That's the reason why we are loading the address of `id`.

General-Purpose Algorithm

1) Compute RHS expr on stack
2) Compute LHS *location* on stack
3) Pop LHS into $t1
4) Pop RHS into $t0
5) Store value $t0 at address $t1

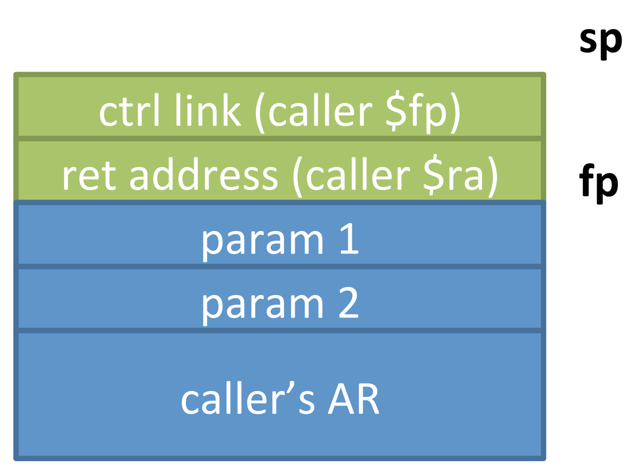


Same Example if id was Global

push 1	`li $t0 1`
	`sw $t0 0($sp)`
	`subu $sp $sp 4`
push 2	`li $t0 2`
	`sw $t0 0($sp)`
	`subu $sp $sp 4`
pop opR	`lw $t1 4($sp)`
	`addu $sp $sp 4`
pop opL	`lw $t0 4($sp)`
	`addu $sp $sp 4`
Do 1+2	`add $t0 $t0 $t1`
push RHS	`sw $t0 0($sp)`
	`subu $sp $sp 4`
push LHS	`la $t0` ~~`-8($fp)`~~ **`id`**
	`sw $t0 0($sp)`
	`subu $sp $sp 4`
pop LHS	`lw $t1 4($sp)`
	`addu $sp $sp 4`
pop RHS	`lw $t0 4($sp)`
	`addu $sp $sp 4`
Do assign	`sw $t0 0($t1)`

<u>General-Purpose Algorithm</u>

1) Compute RHS expr on stack
2) Compute LHS *location* on stack
3) Pop LHS into $t1
4) Pop RHS into $t0
5) Store value $t0 at address $t1



Do We *Need* LHS computation ?

This is a bit much when the LHS is a variable

- We end up doing a single load to find the address, then a store, then a load
- We know a lot of the computation at compile time

Static v Dynamic Computation

Static

- Perform the computation at compile-time

Dynamic

- Perform the computation at runtime

As applied to memory addresses…

- Global variable location
- Local variable
- Field offset

More Complex LHS addresses

Chain of dereferences

    java: a.b.c.d

Array cell address

    arr[1]

    arr[c]

    arr[1][c]

    arr[c][1]

Dereference Computation

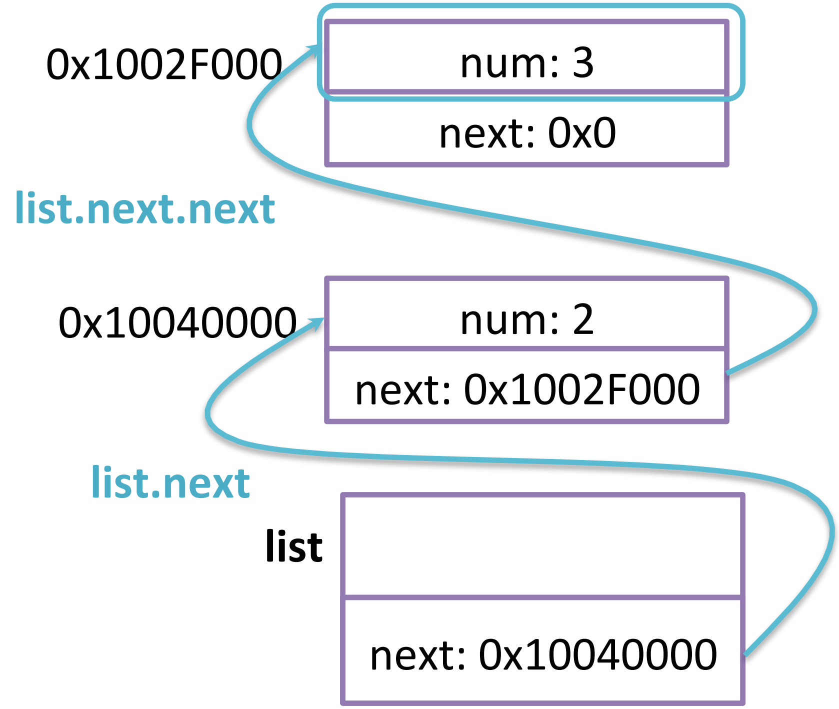
```
struct LinkedList{
    int num;
    struct LinkedList& next;

}
list.next.next.num = list.next.num
```

<u>multi-step code to load this address</u>

<u>multi-step code to load this value</u>

- Get base addr of list
- Get offset to next field
- Load value in next field
- Get offset to next field
- Load value in next field
- Get offset to num field
- Load that address



Control Flow Constructs

Function Calls

Loops

Ifs

Function Call

Two tasks:

- Put argument *values* on the stack (pass-by-value semantics)
- Jump to the callee preamble label
- Bonus 3rd task: save *live* registers
  - (We don't have any in a stack machine)
- Semi-bonus 4th task: retrieve result value

Function Call Example

```
int f(int arg1, int arg2){
  return 2;
}
int main(){
  int a;
  a = f(a,4);
}
```

```
li $t0 4          # push arg 2
sw $t0 0($sp)     #
subu $sp $sp 4    #
lw $t0 -8($fp)    # push arg 1
sw $t0 0($sp)     #
subu $sp $sp 4    #
jal f             # goto f
addu $sp $sp 8    # tear down params
sw $v0 -8($fp)    # retrieve result
```

We Need a New Tool

Control Flow Graph

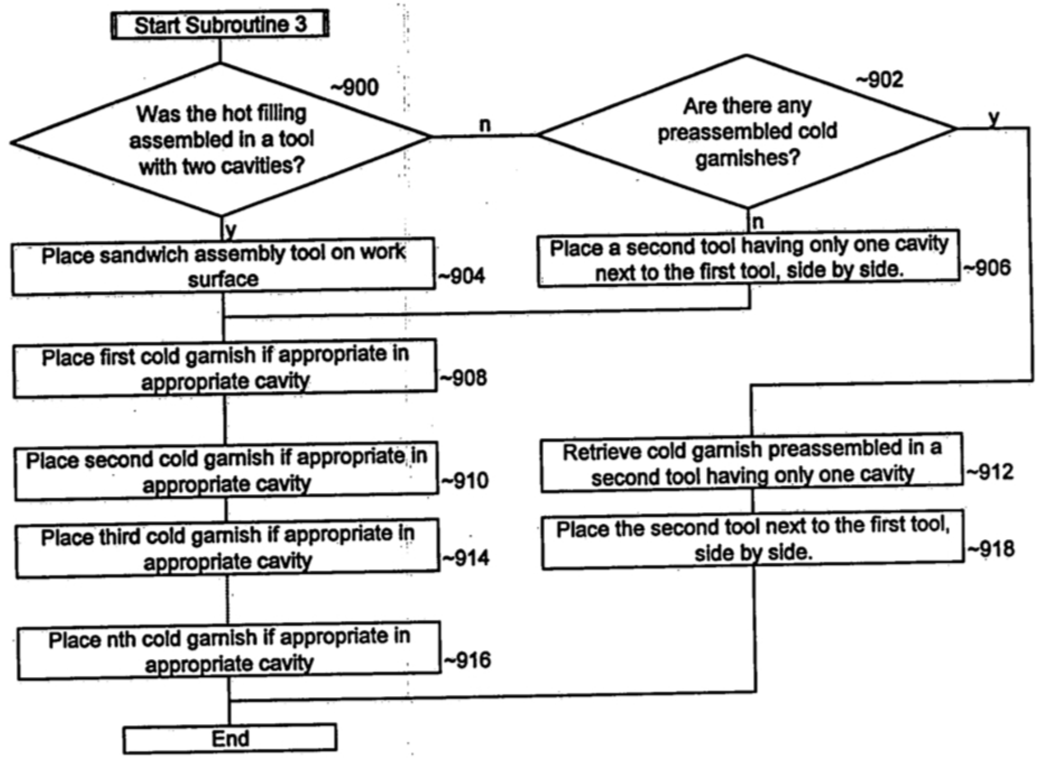
- Important representation for program optimization
- Helpful way to visualize source code

Control Flow Graphs: the Other CFG

Think of a CFG like a flowchart

- Each block is a set of instructions
- Execute the block, decide on next block

Fig. 59

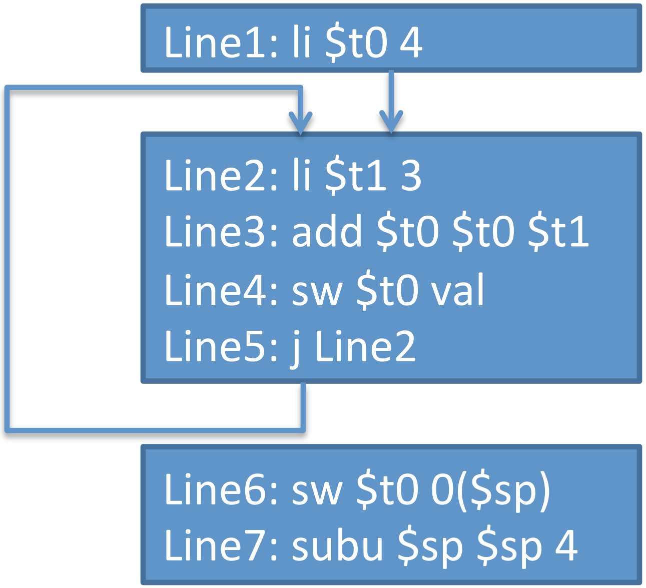


Basic Blocks

Nodes in the CFG

Largest run of instructions that will always be executed in sequence

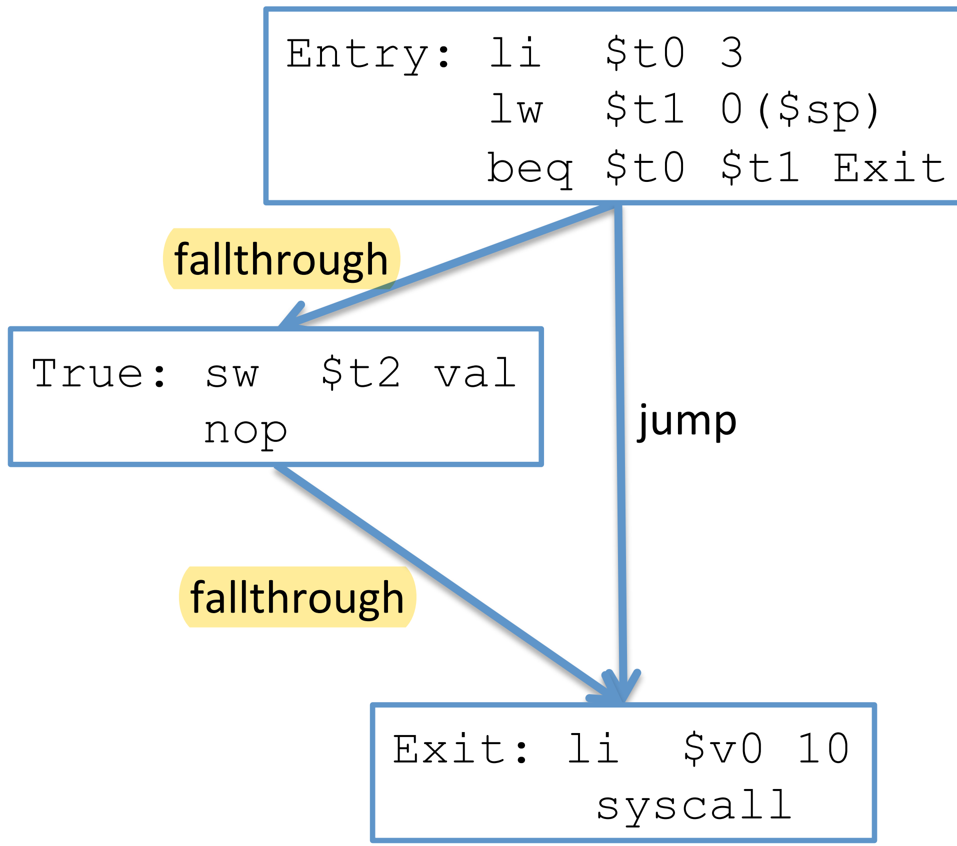
```
Line1: li $t0 4
Line2: li $t1 3
Line3: add $t0 $t0 $t1
Line4: sw $t0 val
Line5: j Line2
Line6: sw $t0 0($sp)
Line7: subu $sp $sp 4
```



Conditional Blocks

Branch instructions cause a node to have multiple out-edges

```
Entry: li   $t0 3
       lw   $t1 0($sp)
       beq $t0 $t1 Exit
 True: sw   $t2 val
       nop
 Exit: li   $v0 10
       syscall
```



Generating If-then Stmts

First, get label for the exit

Generate the head of the if

– Make jumps to the (not-yet placed!) exit label

Generate the true branch

– Write the body of the true node

Place the exit label

If-then Stmts

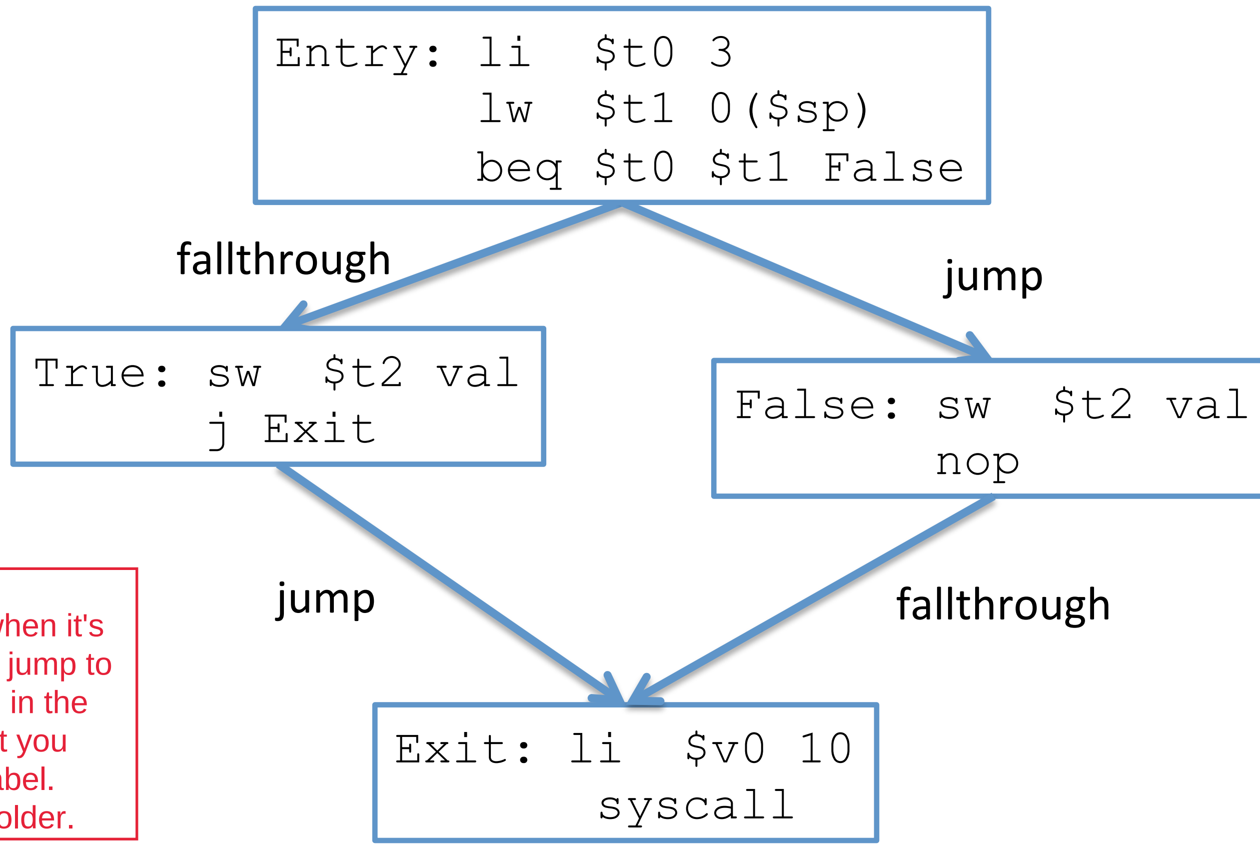
```
…
if (val == 1){
   val = 2;
}
…
```

```
  lw $t0 val          # evaluate condition LHS
  sw $t0 0($sp)       # push onto stack
  subu $sp $sp 4      #
  li $t0 1            # evaluate condition RHS
  sw $t0 0($sp)       # push onto stack
  subu $sp $sp 4      #
  lw $t1 4($sp)       # pop RHS into $t1
  addu $sp $sp 4      #
  lw $t0 4($sp)       # pop LHS into $t0
  addu $sp $sp 4      #
  bne $t0 $t1 L_0     # skip if condition false
  li $t0 2            # Loop true branch
  sw $t0 val
  nop                 # end true branch
L_0:                  # branch successor
  …
```

Conditional Blocks

```
Entry: li  $t0 3
       lw  $t1 0($sp)
       beq $t0 $t1 False
 True: sw  $t2 val
       j Exit
False: sw  $t2 val2
       nop
 Exit: li  $v0 10
       syscall
```

Use fallthrough to reduce the number of jumps needed.



When is nop needed?
Usually not needed. The situation when it's needed is when you have a label to jump to but there's no instruction to execute in the branch (like an empty if branch), but you need some instruction to create a label. Then you can use nop as a place holder.

Generating If-then-Else Stmts

First, get name for the false and exit labels

Generate the head of the if

- Make jumps to the (not-yet placed!) exit and false labels

Generate the true branch

- Write the body of the true node
- Jump to the (not-yet placed!) exit label

Generate the false branch

- Place the false label
- Write the body of the false node

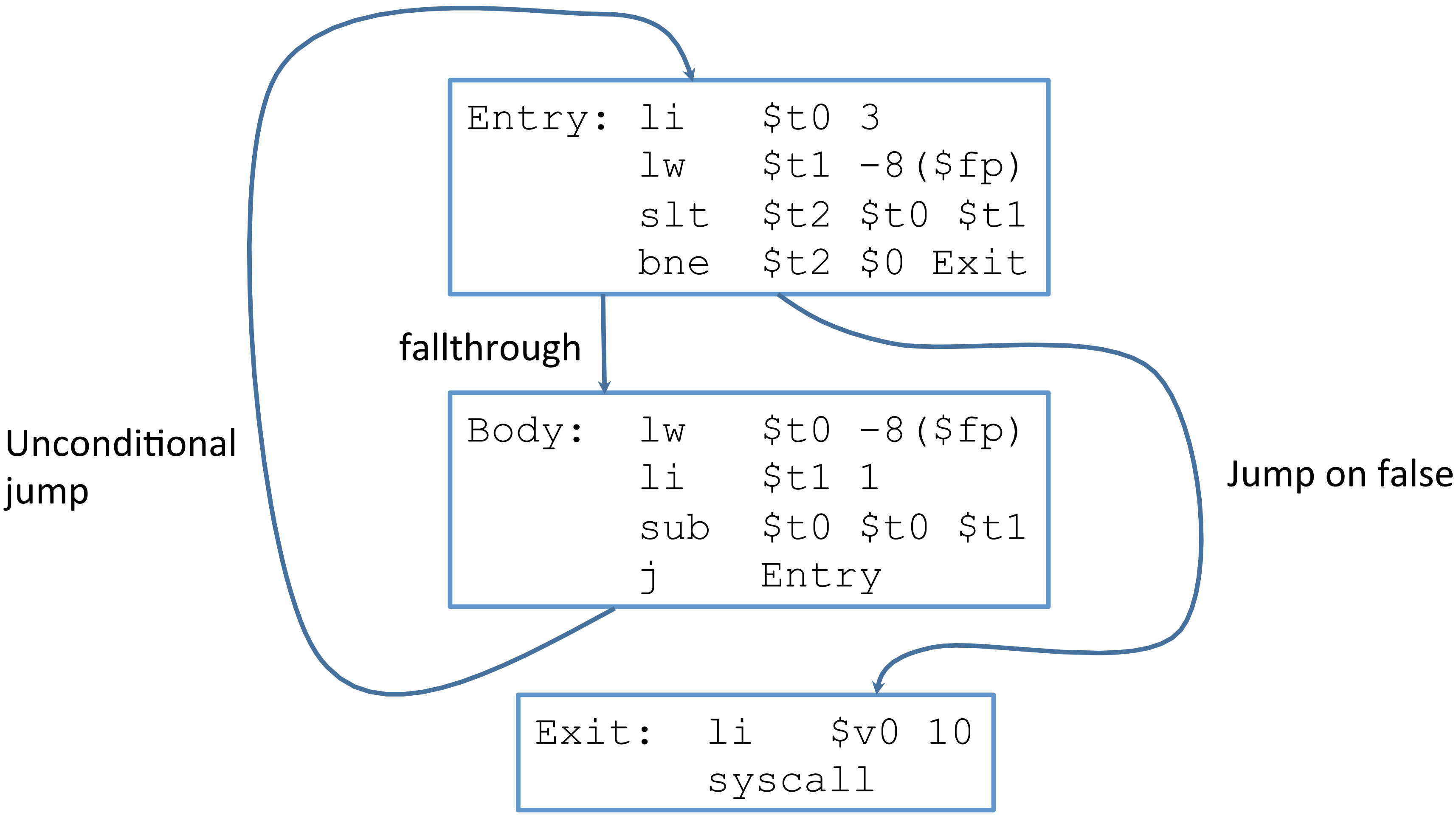
Place the exit label

If-then-Else Stmts

```
…
if (val == 1){
   val = 2;
} else {
   val = 3;
}
…
```

```
  lw $t0 val          # evaluate condition LHS
  sw $t0 0($sp)       # push onto stack
  subu $sp $sp 4      #
  li $t0 1            # evaluate condition RHS
  sw $t0 0($sp)       # push onto stack
  subu $sp $sp 4      #
  lw $t1 4($sp)       # pop RHS into $t1
  addu $sp $sp 4      #
  lw $t0 4($sp)       # pop LHS into $t0
  addu $sp $sp 4      #
  bne $t0 $t1 L_1     # branch if condition
                        false
  li $t0 2            # Loop true branch
  sw $t0 val
  j L_0               # end true branch
L_1:                  # false branch
  …
L_0:                  # branch successor
```

While Loops CFG



Generating While Loops

Very similar to if-then stmts

- Generate a bunch of labels
- Label for the head of the loop
- Label for the successor of the loop

At the end of the loop body

- Unconditionally jump back to the head

Compared to while loop, a do-while loop is simpler:

```
do {
  val = val + 1;
} while (val < 10);
```

Assume val is a global variable.

```
Entry:
# push val to stack
lw $t0 val
sw $t0 0($sp)
addu $sp $sp 4

# push 1 to stack
li $t0 1
sw $t0 0($sp)
addu $sp $sp 4

# pop to $t0
lw $t0 4($sp)
addu $sp $sp 4

# pop to $t1
lw $t1 4($sp)
addu $sp $sp 4

# compute (val + 1)
add $t0 $t0 $t1
sw $t0 0($sp)
subu $sp $sp 4

# assignment
sw $t0 val

# branching
li $t0 val
lw $t1 10
blt $t0 $t1 Entry
```

While Loop

```
while (val == 1){
   val = 2;
}
```

Why jump is bad?
Because jump discards the work done by pipeline. So when encountering a branch, CPU would pretend it's true and execute it in pipeline. If a jump happens, all those efforts are lost. However, if less jump, pipeline can be used more and increases the efficiency.

L0 is the entry, not the body. Need to re-evaluate the body.

```
L_0:
   lw $t0 val          # evaluate condition LHS
   sw $t0 0($sp)       # push onto stack
   subu $sp $sp 4      #
   li $t0 1            # evaluate condition RHS
   sw $t0 0($sp)       # push onto stack
   subu $sp $sp 4      #
   lw $t1 4($sp)       # pop RHS into $t1
   addu $sp $sp 4      #
   lw $t0 4($sp)       # pop LHS into $t0
   addu $sp $sp 4      #
   bne $t0 $t1 L_1     # branch loop end
   li $t0 2            # Loop body
   sw $t0 val
   j L_0               # jump to loop head
L_1:                   # Loop successor
   ...
```

A Note on Conditionals

We lack instructions for branching on most relations

- No “branch if reg1 < reg2”
- Instead we use the slt “set less than”
  - `slt $t2 $t1 $t0`
    - $t2 is 1 when $t1 < $t0
    - $t2 otherwise set to 0

P6 Helper Functions

Generate (opcode, …args…)

- Generate("add", "T0", "T0", "T1")
  - writes out `add $t0, $t0, $t1`
- Versions for fewer args as well

Generate indexed (opcode, "Reg1", "Reg2", offset)

GenPush(reg) / GenPop(reg)

NextLabel() – Gets you a unique label

GenLabel(L) –Places a label

Questions?

Looking forward

- More uses of the CFG
- Program analysis
- Optimization

Homework: see QtSpim resources

QtSpim