Lecture 15

Code generation part 2

Code generation

Hyosu Kim

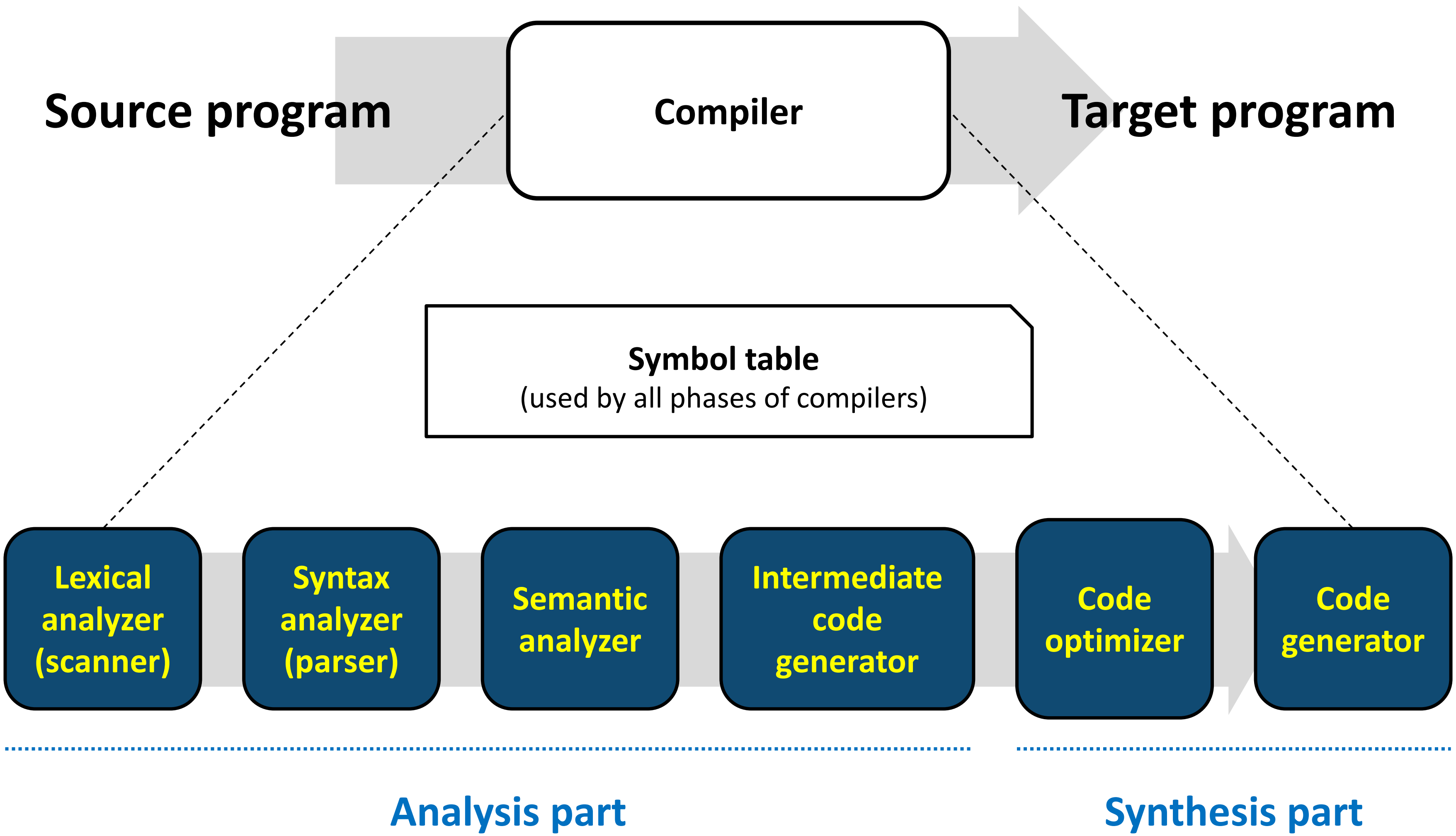
School of Computer Science and Engineering

Chung-Ang University, Seoul, Korea

https://sites.google.com/view/hyosukim

hskimhello@cau.ac.kr, hskim.hello@gmail.com

Overview



Source program
Compiler
Target program
Symbol table
(used by all phases of compilers)
Lexical analyzer (scanner)
Syntax analyzer (parser)
Semantic analyzer
Intermediate code generator
Code optimizer
Code generator
Analysis part
Synthesis part

Code generator

Translates an intermediate representation (e.g., three address code)

into a machine-level code (e.g., assembly code)

Intermediate representation
e.g., TAC

→ **Code generator** →

Machine-level code
e.g., assembly, OPCODE

- Data copy operations
- Arithmetic operations
- Comparison operators
- Control jumps
- **Objects**
  (primitive type variables, arrays, …)
- **Function calls**
- **Parameter passing**

Q. How to support such high-level structures in machine-level code???

- Data copy operations
- Arithmetic operations
- Comparison operators
- Control jumps

Runtime environment!!

Code generator

Translates an intermediate representation (e.g., three address code)

into a machine-level code (e.g., assembly code)

Goal of this stage

- **Choose the appropriate machine instructions for each intermediate representation instruction**
  - **With the use of runtime environments**

- Efficiently allocate finite machine resources (e.g., registers, caches, …)

Assembly languages

Any low-level (machine-level) programing language

Each assembly language is specific to a particular computer architecture

In our class, we shall use a MIPS computer as our target machine

MIPS: Microprocessor without Interlocked Pipelined Stages

- MIPS-based processors were the best selling processors (in the 90s)
- Tons of embedded systems are still using MIPS
- All other mainstream CPU architectures (e.g., x86) also work in a similar way with MIPS

MIPS assembly

Characteristics

- Only load and store instructions access memory

- All other instructions (e.g., arithmetic instructions) use **registers** as operands

- **Register??** dedicated memory locations that
  - Can be accessed quickly
  - Can have computations performed on them
  - But, exist in small quantity
  - **So, it is very important to manage registers properly and efficiently**

- MIPS has 32 general purpose registers

MIPS assembly

Especially, in this class,

- We use a **32-bit** machine architecture
  - Assuming that all objects are **4-byte aligned**
- We use only **two types of registers**
  - **$sp:** a register for storing a stack pointer which points the next location of the top-of-stack
  - **$r0, $r1, …:** registers for storing temporary values
- We assume that there are **an infinite number** of registers ($r0, $r1, ….)
- We translate **each piece of intermediate representations directly to assembly**

MIPS assembly

Basic instructions

- *lw reg1 offset(reg2)*: load 32-bit word from address reg2 + offset into reg1
- *sw reg1 offset(reg2)*: store 32-bit word in reg1 at address reg2 + offset

- *add reg1 reg2 reg3*: reg1 = reg2 + reg3
  - *mul reg1 reg2 reg3*: reg1 = reg2 * reg3
  - *sub reg1 reg2 reg3*: reg1 = reg2 - reg3
  - *div reg1 reg2 reg3*: reg1 = reg2 / reg3

- *seq reg1 reg2 reg3*: reg1 = reg2 == reg3
  - *sne reg1 reg2 reg3*: reg1 = reg2 != reg3
  - *sgt reg1 reg2 reg3*: reg1 = reg2 > reg3
  - *sge reg1 reg2 reg3*: reg1 = reg2 >= reg3
  - *slt reg1 reg2 reg3*: reg1 = reg2 < reg3
  - *sle reg1 reg2 reg3*: reg1 = reg2 <= reg3

MIPS assembly

Basic instructions

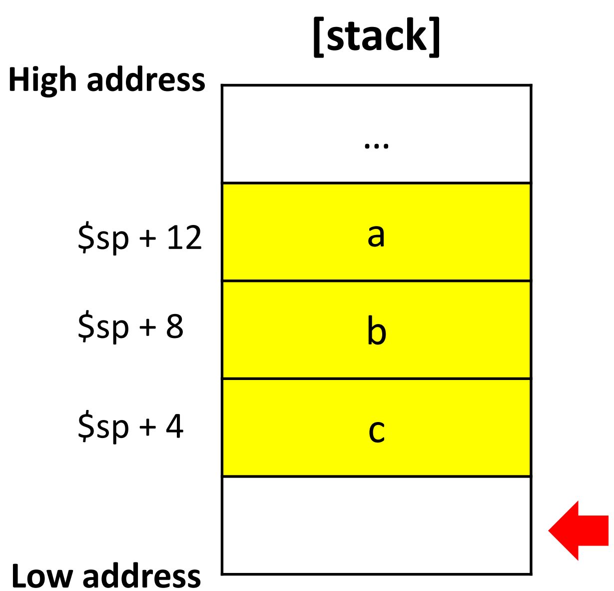
- *li reg1 immediate* (e.g., constant numbers): reg1 = immediate
- *addi (subi, muli, or divi) reg1 reg2 immediate*: reg1 = reg2 + immediate
- *seqi (snei, sgti, sgei, slti, or slei) reg1 reg2 immediate*: reg1 = reg2 == immediate

Code generation for simple operations

Let's suppose that we have a TAC: "a = b + c"

When we execute the TAC, the variables (a, b, and c) already exist in the stack

Because they are the local variables used in the currently-running function



$sp points the next location of the top-of-stack

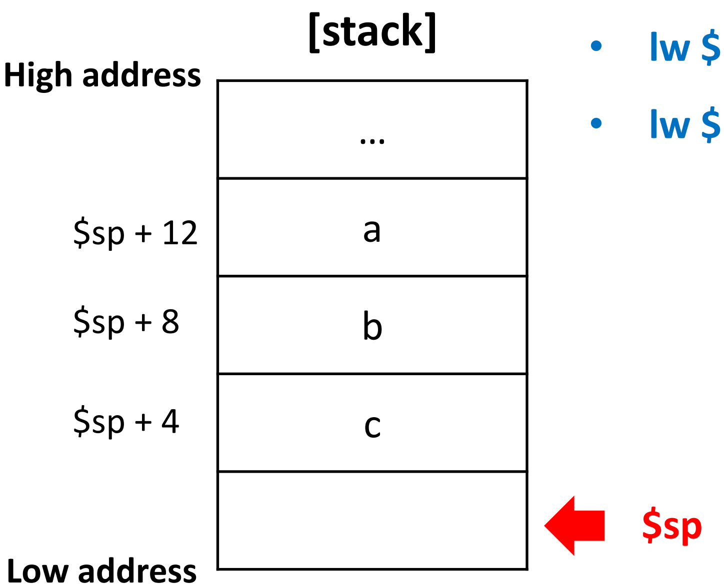
The location of each variable relative to $sp is kept in a symbol table

Code generation for simple operations

Let's suppose that we have a TAC: "a = b + c"

When we execute the TAC, the variables (a, b, and c) already exist in the stack

Because they are the local variables used in the currently-running function



Step1: Load variables from memory to registers

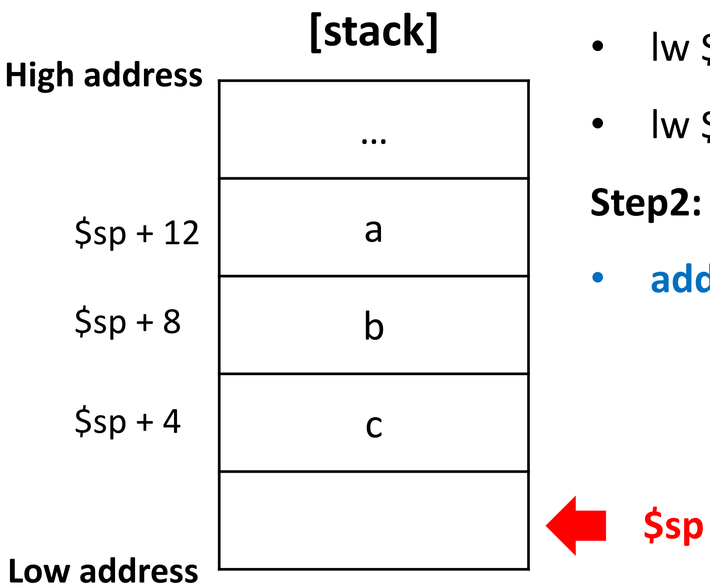
- lw $r0 8($sp)
- lw $r1 4($sp)

Code generation for simple operations

Let's suppose that we have a TAC: "a = b + c"

When we execute the TAC, the variables (a, b, and c) already exist in the stack

Because they are the local variables used in the currently-running function



Step1: Load variables from memory to registers

- lw $r0 8($sp)
- lw $r1 4($sp)

Step2: Do the add operation with registers

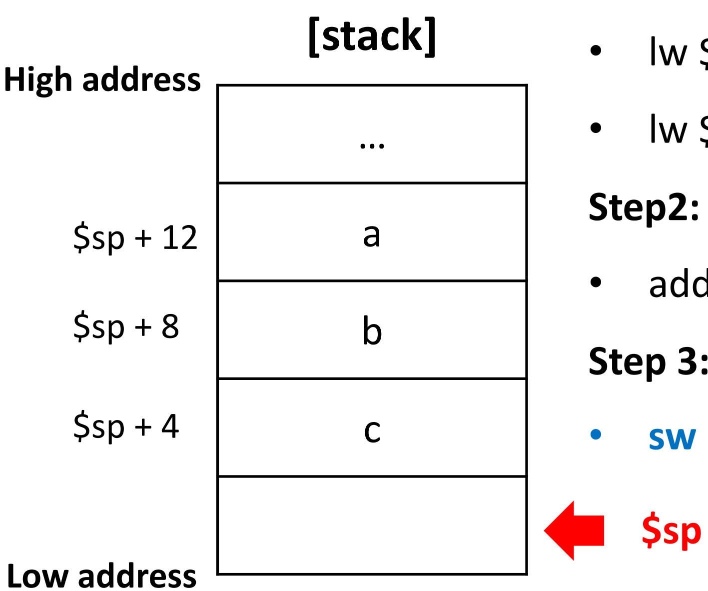
- **add $r2 $r0 $r1**

Code generation for simple operations

Let's suppose that we have a TAC: “a = b + c”

When we execute the TAC, the variables (a, b, and c) already exist in the stack

Because they are the local variables used in the currently-running function



Step1: Load variables from memory to registers

- lw $r0 8($sp)
- lw $r1 4($sp)

Step2: Do the add operation with registers

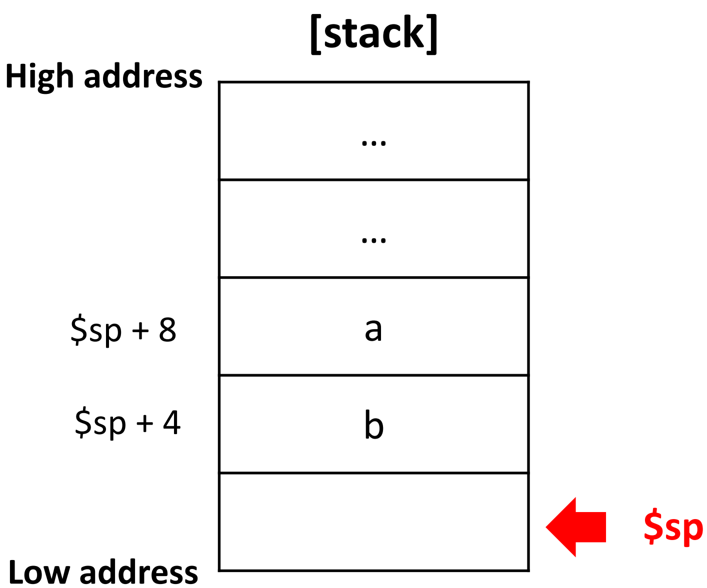
- add $r2 $r0 $r1

Step 3: Store the computation result to memory

- **sw $r2 12($sp)**

Code generation for simple operations

Let's suppose that we have a TAC: "a = b < 3"



Step1: Load variables from memory to registers

- lw $r0 4($sp)

Step2: Do the comparison operation with registers and immediate

- **slti $r1 $r0 3**

Step 3: Store the computation result to memory

- sw $r1 8($sp)

Code generation for conditional jumps

New MIPS assembly instructions for flow control

- ***j label***: unconditional jump to label (goto label)

- ***beq reg1 reg2 label***: if reg1 == reg2 goto label
  - Instead of beq, we can use bne (!=), bgt (>), bge (>=), blt (<), ble (<=)

- ***beqz reg1 label***: if reg1 == 0 goto label
  - Instead of beqz, we can use bnez (!= 0)

- ***beqi reg1 immediate label***: if reg1 == immediate goto label
  - Instead of beqi, we can use bnei (!=), bgti (>), bgei (>=), blti (<), blei (<=)

Code generation for conditional jump

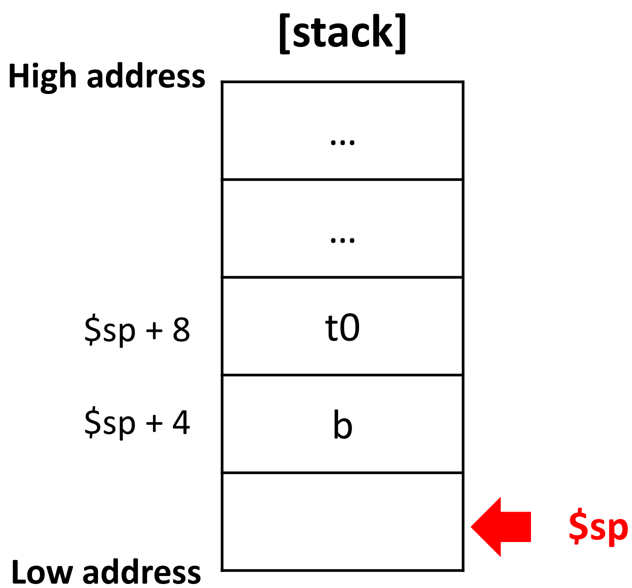
Example

[TAC]

t0 = b < 3

if t0 goto L0

L0:



[Assembly]

```
lw $r0 4($sp)
slti $r1 $r0 3
sw $r1 8($sp)

lw $r0 8($sp)
bnez $r0 L0
L0:
```

Code generation for conditional jump

Practice

[TAC]

L0:

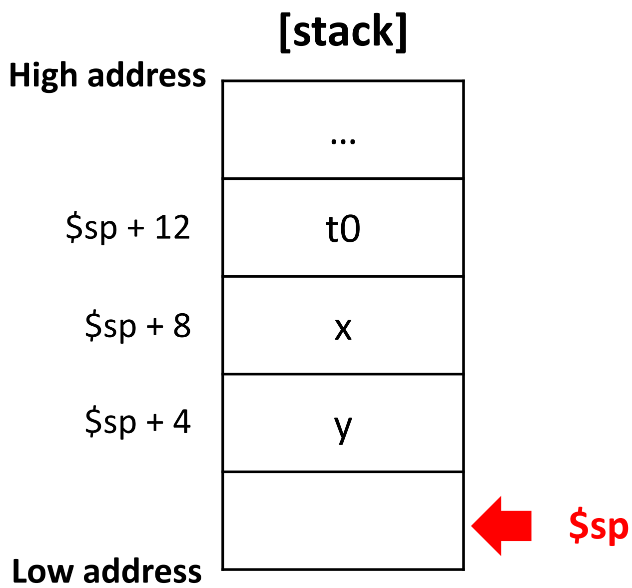
  t0 = x < y

  itnot t0 goto L1

  x = x + 1

  goto L0

L1:

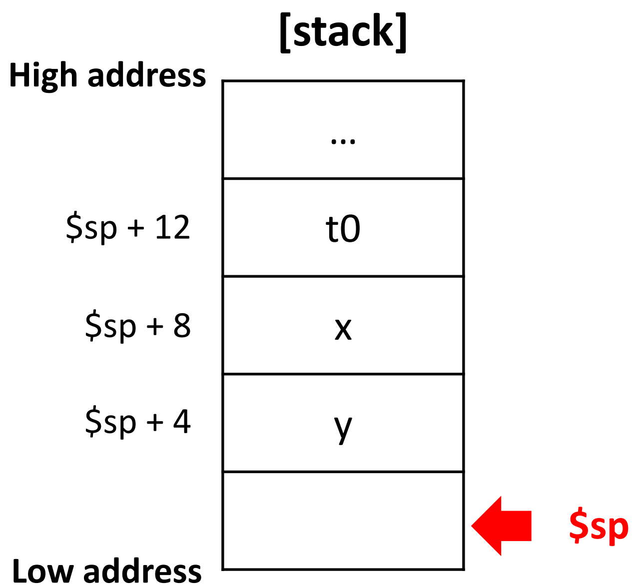


Code generation for conditional jump

Practice

[TAC]

```
L0:
    t0 = x < y
    itnot t0 goto L1
    x = x + 1
    goto L0
L1:
```



[Assembly]

```
L0:
lw $r0 8($sp)
lw $r1 4($sp)
slt $r2 $r0 $r1
sw $r2 12($sp)

lw $r0 12($sp)
beqz $r0 L1

lw $r0 8($sp)
addi $r1 $r0 1
sw $r1 8($sp)
j L0

L1:
```

Code generation for function calls

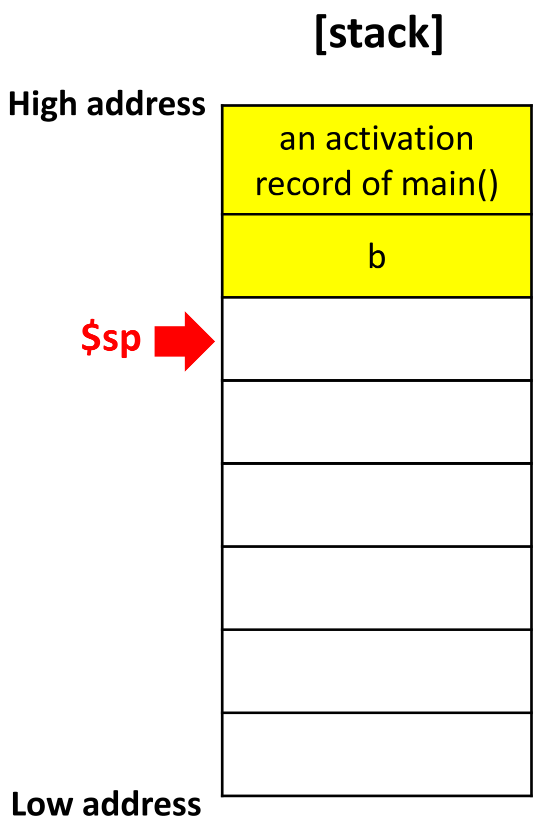
Before foo() is called

- An activation record of main() is stored in a stack

[TAC]

```
begin foo
    t0 = a + 1
    return t0
end foo

begin main
    …
    param 3
    b = call foo, 1
    …
end main
```



Code generation for function calls

When a function foo() is called in a function main(),

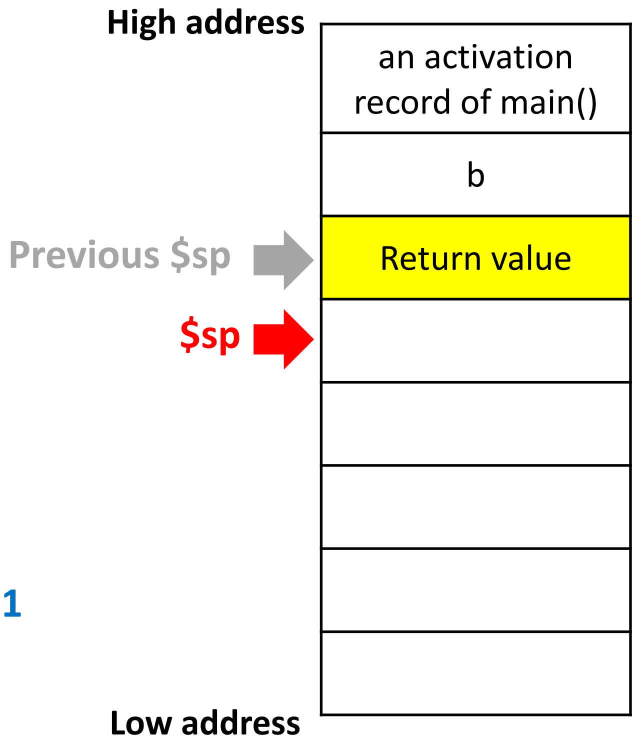
- Store an activation record of foo() into the stack

[TAC]

```
begin foo
   t0 = a + 1
   return t0
end foo

begin main
   …
   param 3
   b = call foo, 1
   …
end main
```

[stack]



Create a space for the return value of foo()

- subi $sp $sp 4

Code generation for function calls

When a function foo() is called in a function main(),

- Store an activation record of foo() into the stack

[TAC]

begin foo

  t0 = a + 1

  return t0

end foo

begin main

  …

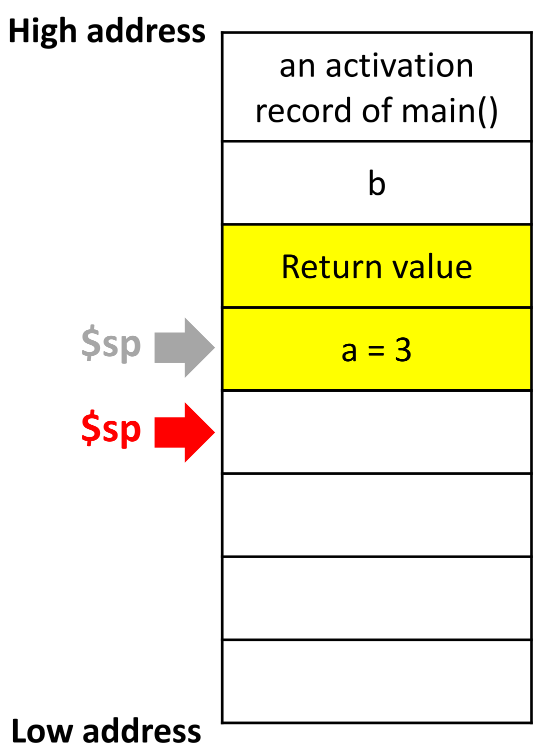
  param 3

  b = call foo, 1

  …

end main

[stack]



Create a space for the return value of foo()

- subi $sp $sp 4

Store input parameters

- li $r0 3
- sw $r0 0($sp)
- subi $sp $sp 4

Code generation for function calls

When a function foo() is called in a function main(),

- Store an activation record of foo() into the stack

[TAC]

begin foo

  t0 = a + 1

  return t0

end foo

begin main

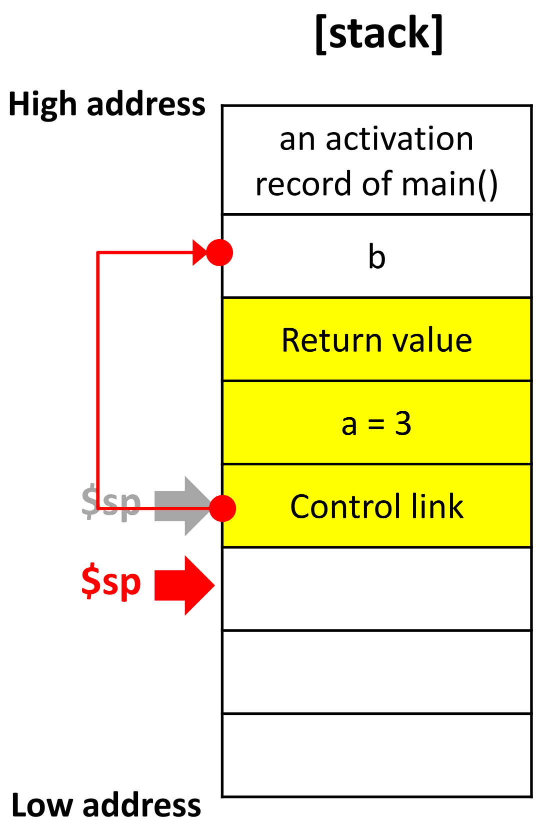
  …

  param 3

  b = call foo, 1

  …

end main



Create a space for the return value of foo()

- subi $sp $sp 4

Store input parameters

- li $r0 3
- sw $r0 0($sp)
- subi $sp $sp 4

Store control link

(the start address of an activation record of main())

- addi $r0 $sp 12
- sw $r0 0($sp)
- subi $sp $sp 4

Code generation for function calls

When a function foo() is called in a function main(),

- Store an activation record of foo() into the stack

[TAC]

begin foo

t0 = a + 1

return t0

end foo

begin main

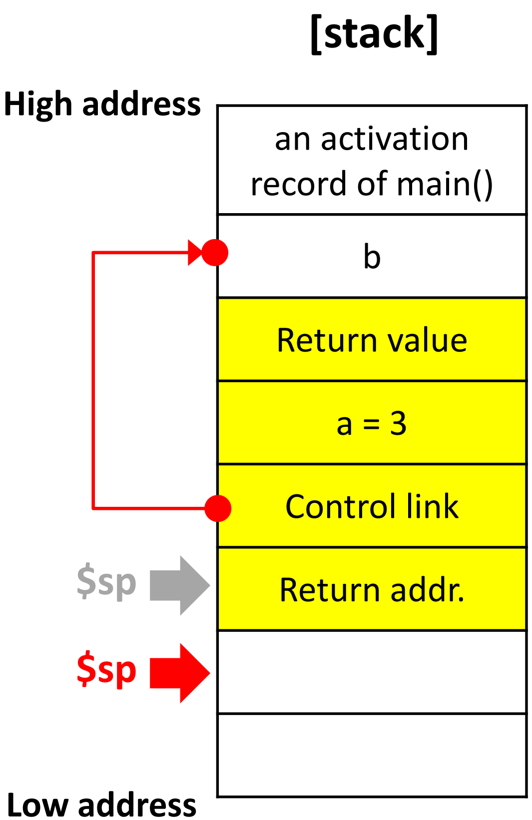
...

param 3

b = call foo, 1

...

end main



Store current machine status

(e.g., return address)

- sw **$ra** 0($sp)

  ($ra is a register for return addresses)

- subi $sp $sp 4

Code generation for function calls

When a function foo() is called in a function main(),

- Store an activation record of foo() into the stack

[TAC]

begin foo

    t0 = a + 1

    return t0

end foo

begin main

    …

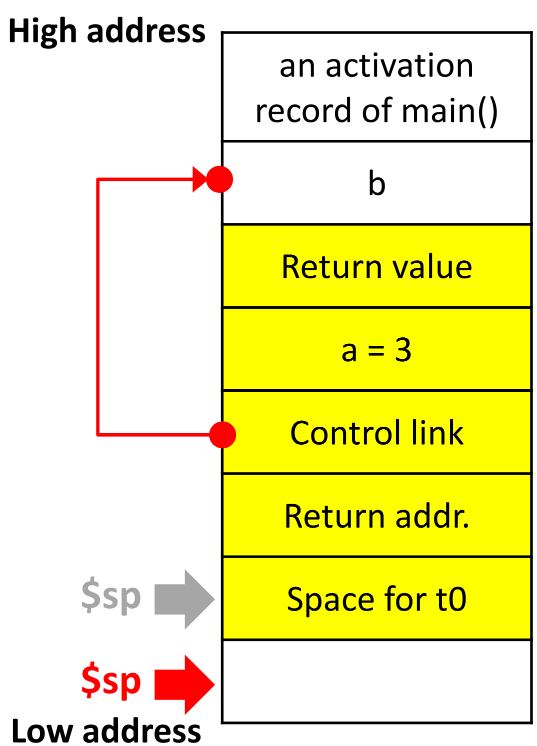
    param 3

    b = call foo, 1

    …

end main

[stack]



Store current machine status

(e.g., return address)

- sw $ra 0($sp)

  ($ra is a register for return addresses)

- subi $sp $sp 4

Create a space for the local variables of foo()

(The information about which variable will be used in foo() is stored in a symbol table)

- subi $sp $sp 4

Code generation for function calls

When a function foo() is executed,

[TAC]

begin foo

t0 = a + 1

return t0

end foo

begin main

…

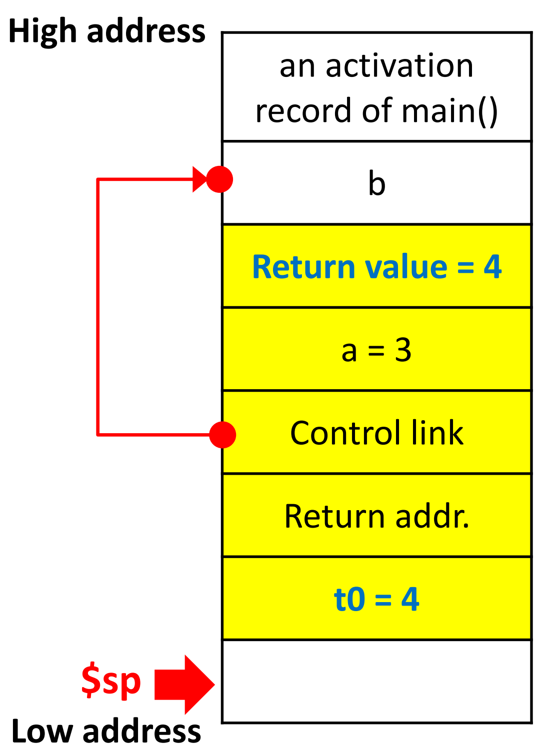
param 3

b = call foo, 1

…

end main

[stack]



Compute t0 = a + 1

- lw $r0 16($sp)
- addi $r1 $r0 1
- sw $r1 4($sp)

Store the return value (t0)

- lw $r0 4($sp)
- sw $r0 20($sp)

Code generation for function calls

When the execution of a function foo() is completed

[TAC]

begin foo

t0 = a + 1

return t0

end foo

begin main

…

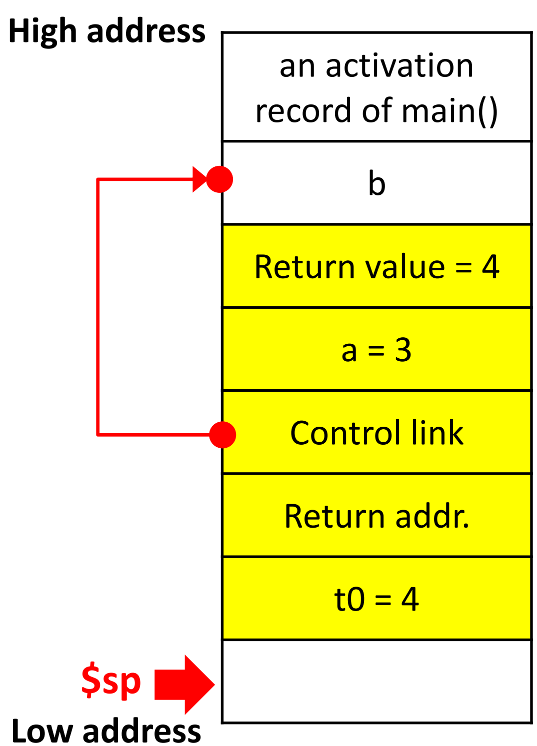
param 3

b = call foo, 1

…

end main

[stack]



Restore machine status (e.g., return address)

- lw $ra 8($sp)

Code generation for function calls

When the execution of a function foo() is completed

[TAC]

begin foo

    t0 = a + 1

    return t0

end foo

begin main

    …

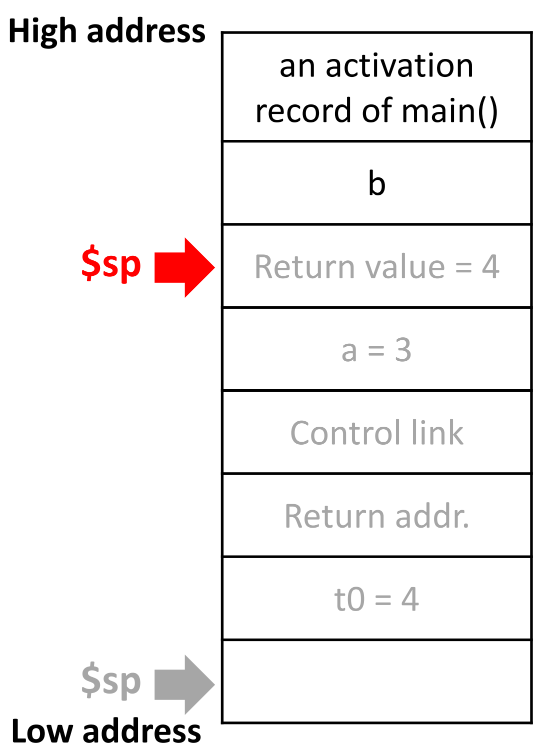
    param 3

    b = call foo, 1

    …

end main

[stack]



Restore machine status (e.g., return address)

- lw $ra 8($sp)

The activation record of foo() is popped out

(Just move $sp to the address

to which the control link points)

- lw $r0 12($sp)
- subi $sp $r0 4

Code generation for function calls

When the execution of a function foo() is completed

[TAC]

begin foo

  t0 = a + 1

  return t0

end foo

begin main

  …

  param 3

  b = call foo, 1

  …

end main

[stack]

High address	an activation record of main()
	b = 4
$sp →	Return value = 4
	a = 3
	Control link
	Return addr.
	Space for t0 = 4
Low address	

Restore machine status (e.g., return address)

- lw $ra 8($sp)

The activation record of foo() is popped out

(Just move $sp to the address

 to which the control link points)

- lw $r0 12($sp)
- subi $sp $r0 4

Go back to the return address and do work!

- **jr $ra** (Jump to the address in $ra)
- lw $r0 0($sp) (copy the return value)
- sw $r0 4($sp)

Problems in our assumptions

Especially, in this class,

- We assume that there are an infinite number of registers ($r0, $r1, ….)

  It is not realistic

  - In practice, we have a limited number of registers

- We translate each piece of intermediate representations directly to assembly

  It allows simplicity in generating assembly, but compromising efficiency

  - It issues unnecessary loads and stores

Code generator

Translates an intermediate representation (e.g., three address code)

into a machine-level code (e.g., assembly code)

Intermediate representation
e.g., TAC

→ **Code generator** →

Machine-level code
e.g., assembly, OPCODE

Goal of this stage

- Choose the appropriate machine instructions for each intermediate representation instruction
  - With the use of runtime environments

- **Efficiently allocate finite machine resources (e.g., registers, caches, …)**

Through a better register allocation!!