Languages and Compilers (SProg og Oversættere)

Lecture 18
Low Level Code Generation

Bent Thomsen

Department of Computer Science

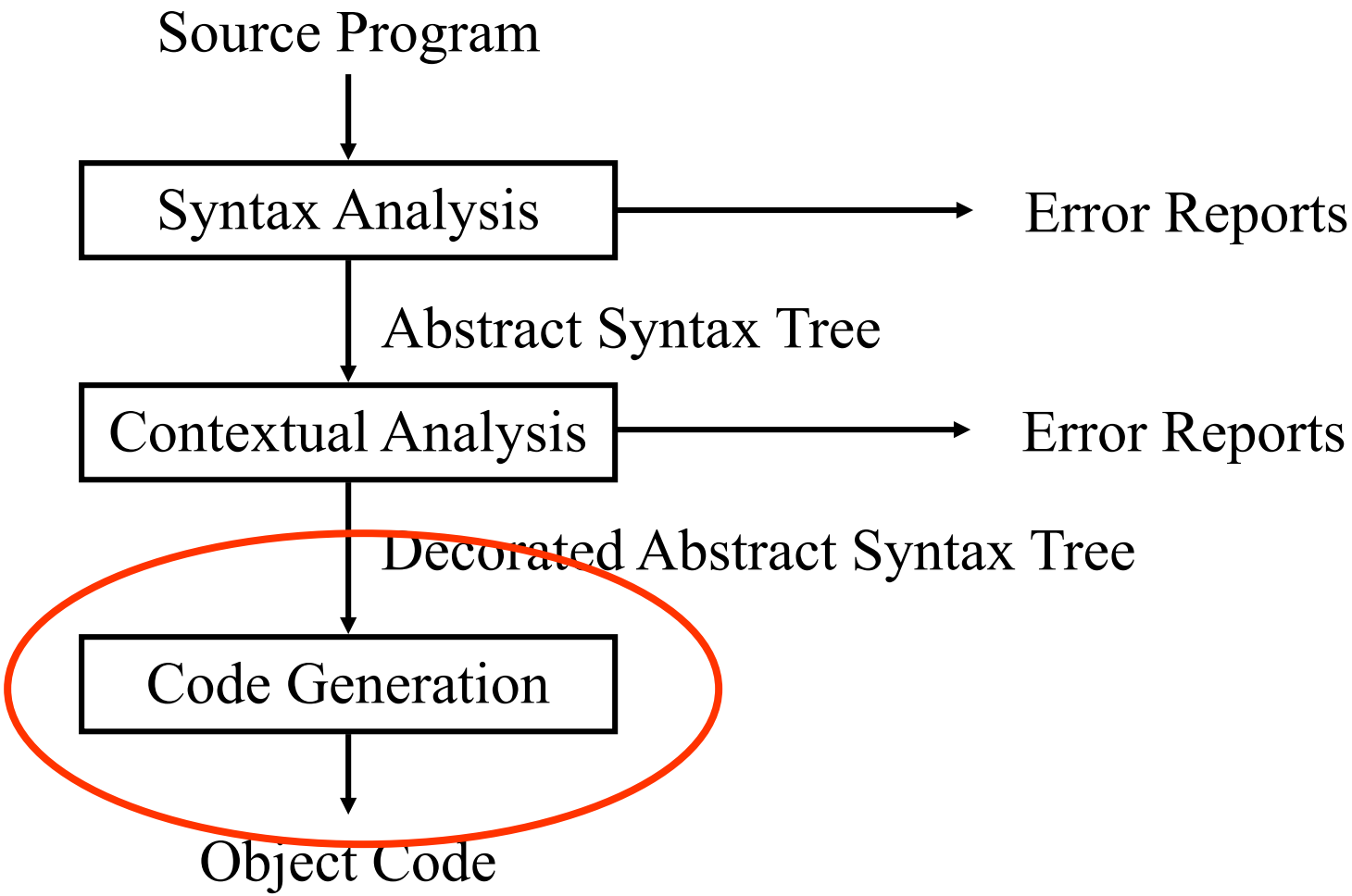
Aalborg University

Learning goals

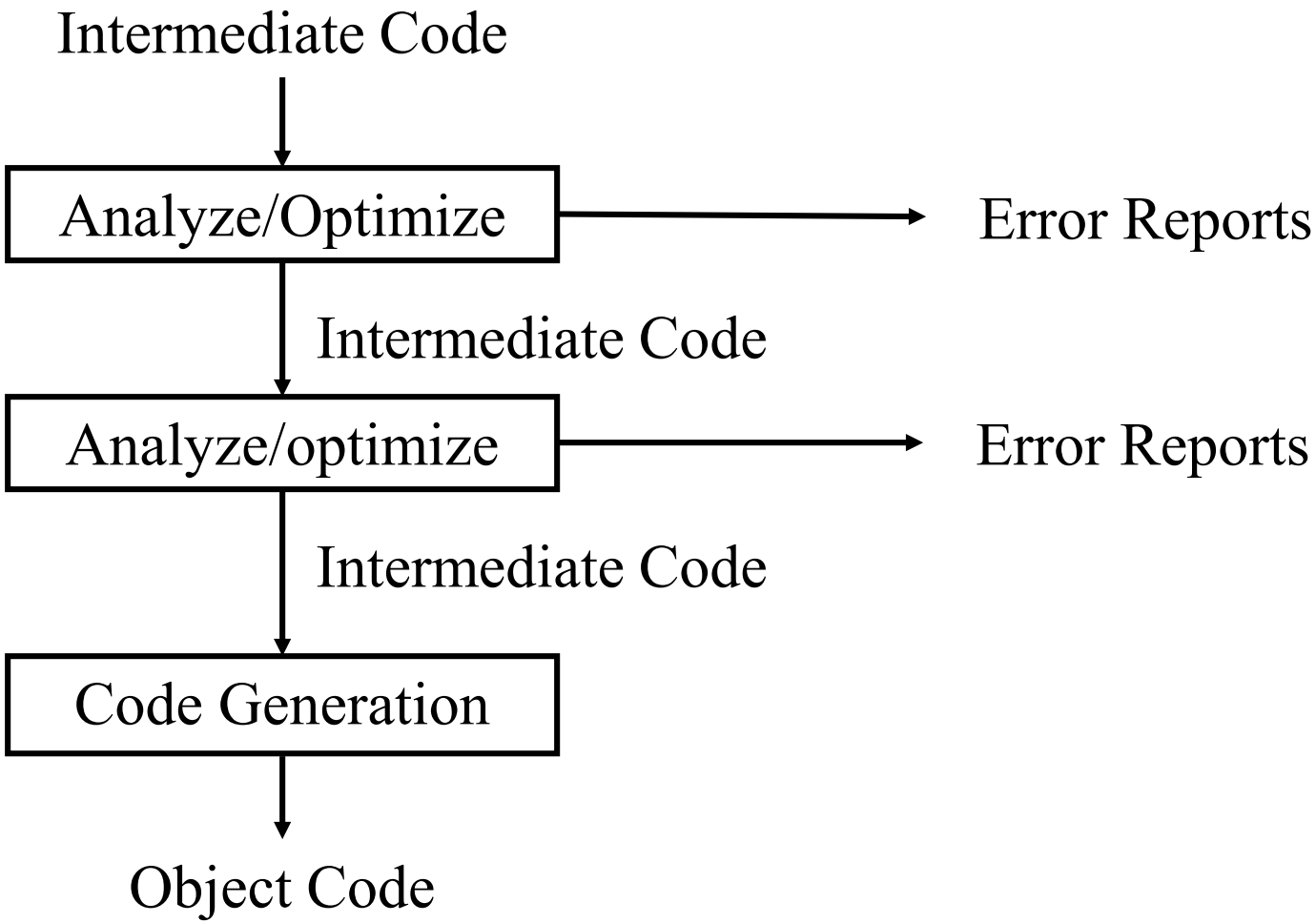
- Understand issues such as
  - code selection
  - storage allocation
  - register allocation
  - code scheduling

  for low level code generation.
- Understand different approaches to low level code generation:
  - Code generation from AST via visitor
  - Code generation by tree-rewrite and pattern matching
  - Code generation from IR

The "Phases" of a Compiler



The "Phases" of a Compiler



Intermediate Representations

- Abstract Syntax Tree
  - Convenient for semantic analysis phases
  - We can generate code directly from the AST, but...
  - What about multiple target architectures?
    - Remember n * m vs. n + m
- Intermediate Representation
  - "Neutral" architecture
  - Easy to translate to native code
  - Can abstracts away complicated runtime issues
    - Stack Frame Management
    - Memory Management
    - Register Allocation

Issues in Code Generation

- Code Selection:

  Deciding which sequence of target machine instructions will be used to implement each phrase in the source language.

- Storage Allocation

  Deciding the storage address for each variable in the source program. (static allocation, stack allocation etc.)

- Register Allocation (for register-based machines)

  How to use registers efficiently to store intermediate results.

- Code Scheduling

  The order in which the generated instructions are executed

Code generation from AST summary

- Idea from Brown & Watt
- Create code templates inductively
  - There may be special case templates generating equivalent, but more efficient code
  - Keep in mind what goes on at compile time
    - AST traversal order
  - Keep in mind what goes on at run time
    - Control flow order
- Use visitors (or composit or functional) pattern to walk the AST recursively emitting code as you go along
- Low level VM, called Triangle VM, with direct addressing and storage allocation

Developing a Code Generator "Visitor"

Phrase Class	visitor method	Behavior of the visitor method
Program	visitProgram	generate code as specified by *run*[P]
Command	visit…Command	generate code as specified by *execute*[C]
Expression	visit…Expression	generate code as specified by *evaluate*[E]
V-name	visit…Vname	Return "entity description" for the visited variable or constant name.
Declaration	visit…Declaration	generate code as specified by *elaborate*[D]
Type-Den	visit…TypeDen	return the size of the type

Example from Brown&Watt chapter 7, p. 260- 280, translating miniTriangle to TAM, a stack based VM with explicit addressing and storage allocation

Developing a Code Generator "Visitor"

evaluate [IL] =

LOADL *v* where *v* is the integer value of IL

```
/* Expressions */
public Object visitIntegerExpression (
            IntegerExpression expr,Object arg) {
   short v = valuation(expr.IL.spelling);
   emit(Instruction.LOADLop, 0, 0, v);
   return null;
}

public short valuation(String s) {
   ... convert string to integer value ...
}
```

Developing a Code Generator "Visitor"

evaluate [E1 O E2] =
    evaluate [E1]
    evaluate [E2]
    CALL *p* where *p* is the address of routine for O

```
public Object visitBinaryExpression (
            BinaryExpression expr,Object arg) {
   expr.E1.visit(this,arg);
   expr.E2.visit(this,arg);
   short p = address for expr.O operation
   emit(Instruction.CALLop,
      Instruction.SBr,
      Instruction.PBr, p);
   return null;
}
```

Remaining expression visitors are developed in a similar way.

Developing a Code Generator "Visitor"

execute [V := E] =
    evaluate [E]
    assign [V]

```
/* Generating code for commands */

public Object visitAssignCommand(
                  AssignCommand com,Object arg) {
   com.E.visit(this,arg);
   RuntimeEntity entity =
      (RuntimeEntity) com.V.visit(this,null);
   short d = entity.address;
   emit(Instruction.STOREop,Com.V.size,d);
   return null;
}
```

Developing a Code Generator "Visitor"

execute [C1 ; C2] =
    execute[C1]
    execute[C2]

```
public Object visitSequentialCommand(
                SequentialCommand com,Object arg) {
   com.C1.visit(this,arg);
   com.C2.visit(this,arg);
   return null;
}
```

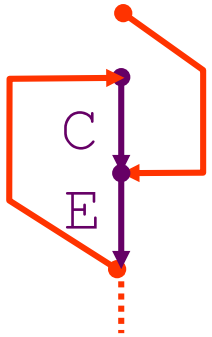
- IfCommand and WhileCommand: complications with jumps
- LetCommand is more complex: memory allocation and addresses

Control Structures

We have yet to discuss generation for IfCommand and WhileCommand

execute [**while** E **do** C] =

```
        JUMP h
g:      execute [C]
h:      evaluate[E]
        JUMPIF(1) g
```



A complication is the generation of the correct addresses for the jump instructions.

We can determine the address of the instructions by incrementing a counter while emitting instructions.

Backwards jumps are easy but forward jumps are harder.
Q: why?

Control Structures

Backwards jumps are easy:

The "address" of the target has already been generated and is known

Forward jumps are harder:

When the jump is generated the target is not yet generated so its address is not (yet) known.

There is a solution which is known as **backpatching**

1) Emit jump with "dummy" address (e.g. simply 0).
2) Remember the address where the jump instruction occurred.
3) When the target label is reached, go back and patch the jump instruction.

Backpatching Example

```
public Object WhileCommand (
            WhileCommand com,Object arg) {
   short j = nextInstrAddr;
   emit(Instruction.JUMPop, 0,
        Instruction.CBr,0);
   short g = nextInstrAddr;
   com.C.visit(this,arg);
   short h = nextInstrAddr;
   code[j].d = h;
   com.E.visit(this,arg);
   emit(Instruction.JUMPIFop, 1,
        Instruction.CBr,g);
   return null;
}
```

dummy address

backpatch

execute [**while** E **do** C] =

```
        JUMP  h
   g:  execute [C]
   h:  evaluate[E]
       JUMPIF(1)  g
```

Static Storage Allocation: In the Code Generator

```
public Object visit...Command(
    ...Command com, Object arg) {
    short gs = shortValueOf(arg);
    generate code as specified by execute[com]
    return null;
}
public Object visit...Expression(
    ...Expression expr, Object arg) {
    short gs = shortValueOf(arg);
    generate code as specified by evaluate[expr]
    return new Short(size of expr result);
}
public Object visit...Declaration(
    ...Declaration dec, Object arg) {
    short gs = shortValueOf(arg);
    generate code as specified by elaborate[dec]
    return new Short(amount of extra allocated by dec);
}
```

Routines

We call the assembly language equivalent of procedures “routines”.

What are routines? Unlike procedures/functions in higher level languages. They are not directly supported by language constructs. Instead they are modeled in terms of how to use the low-level machine to “emulate” procedures.

What behavior needs to be “emulated”?

- Calling a routine and returning to the caller after completion.
- Passing arguments to a called routine
- Returning a result from a routine
- Local and non-local variables.

Code Generation for Procedures and Functions

We extend Mini Triangle with procedures:

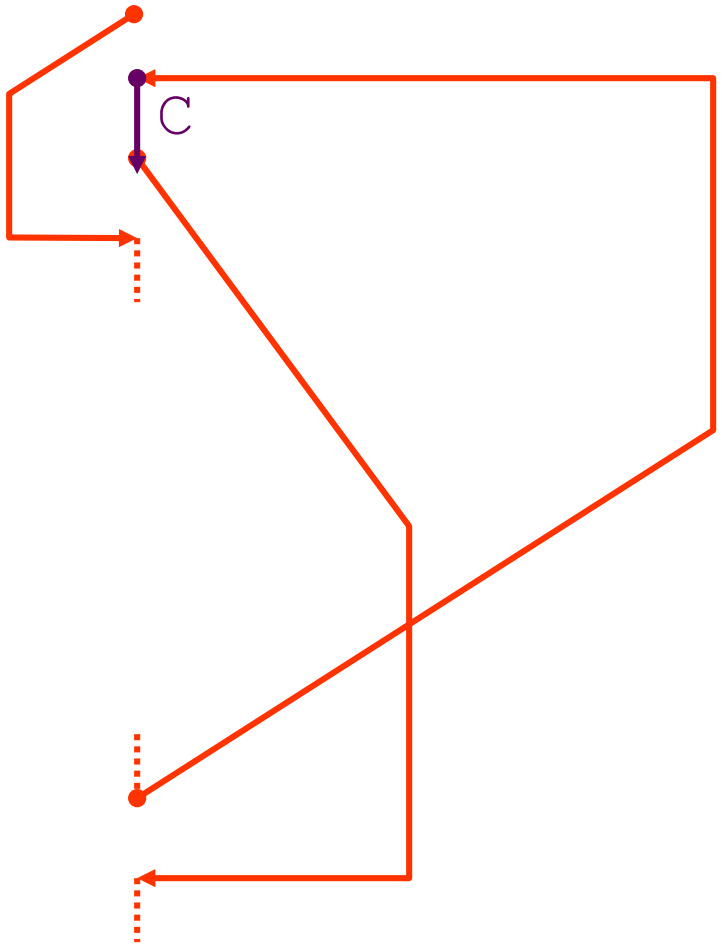
```
Declaration
   ::= ...
     | proc Identifier ( ) ~ Command
Command
   ::= ...
     | Identifier ( )
```

First , we will only consider global procedures (with no arguments).

Code Template: Global Procedure

elaborate [**proc** I () ~ C] =

```
        JUMP g
    e:  execute [C]
        RETURN(0) 0
    g:
```



execute [I ()] =

```
        CALL(SB) e
```

Routines

- Transferring control to and from routine:
  Most low-level processors have `CALL` and `RETURN` for transferring control from caller to callee and back.
- Transmitting arguments and return values:
  Caller and callee must agree on a method to transfer argument and return values.
  => This is called the "routine protocol"

There are many possible ways to pass argument and return values.
=> A routine protocol is like a **"contract" between the caller and the callee**.

The routine protocol is often dictated by the operating system.

Routine Protocol Examples

The routine protocol depends on the machine architecture (e.g. stack machine versus register machine).

Example 1: A possible routine protocol for a RM

- Passing of arguments:
  first argument in R1, second argument in R2, etc.
- Passing of return value:
  return the result (if any) in R0

Note: this example is simplistic:

- What if more arguments than registers?
- What if the representation of an argument is larger than can be stored in a register.

For RM protocols, the protocol usually also specifies who (caller or callee) is responsible for saving contents of registers.

Routine Protocol Examples

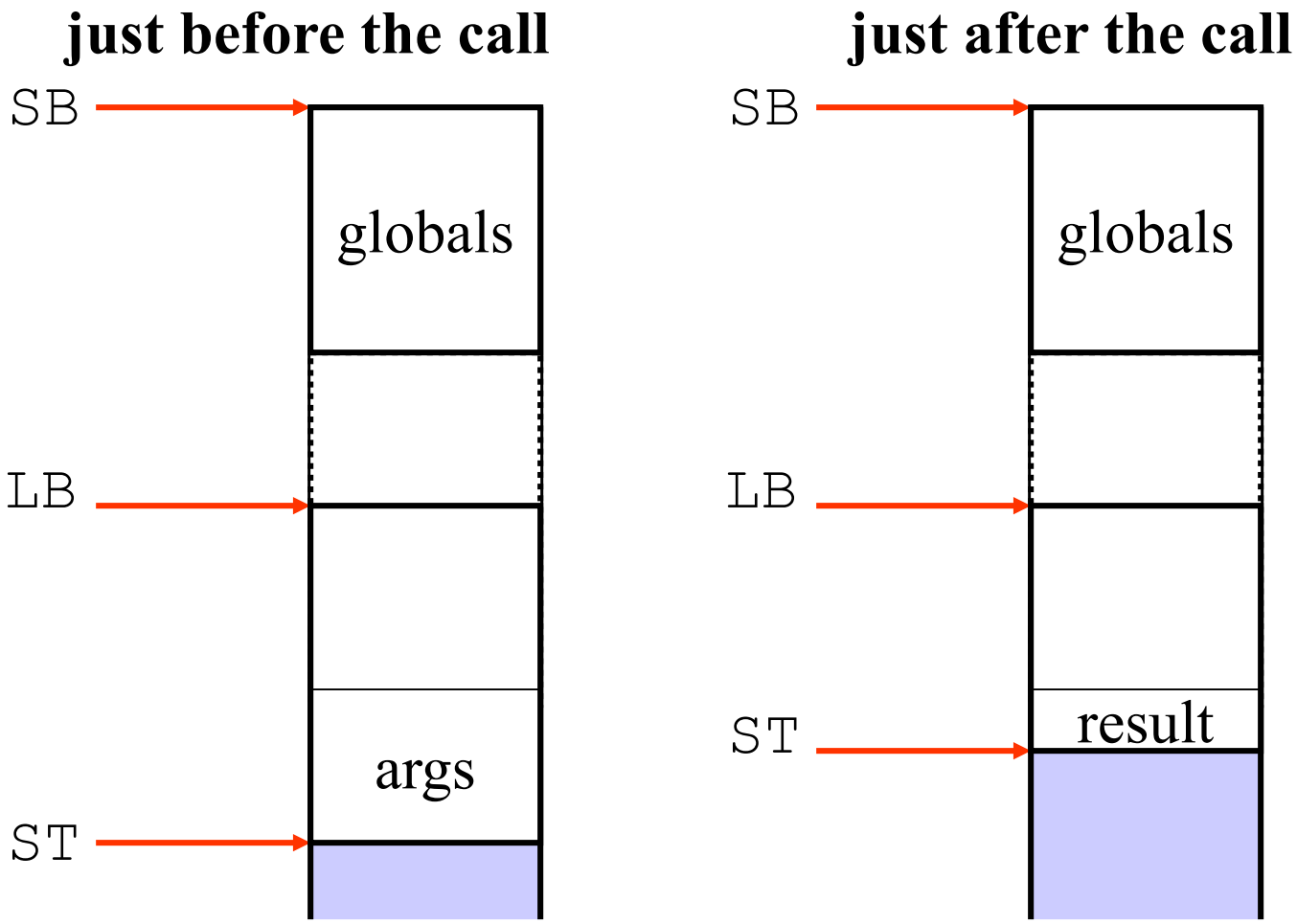
Example 2: A possible routine protocol for a stack machine

- Passing of arguments:
  pass arguments on the top of the stack.
- Passing of return value:
  leave the return value on the stack top, in place of the arguments.

Note: this protocol puts no boundary on the number of arguments and the size of the arguments.

Most micro-processors, have registers as well as a stack. Such "mixed" machines also often use a protocol like this one.

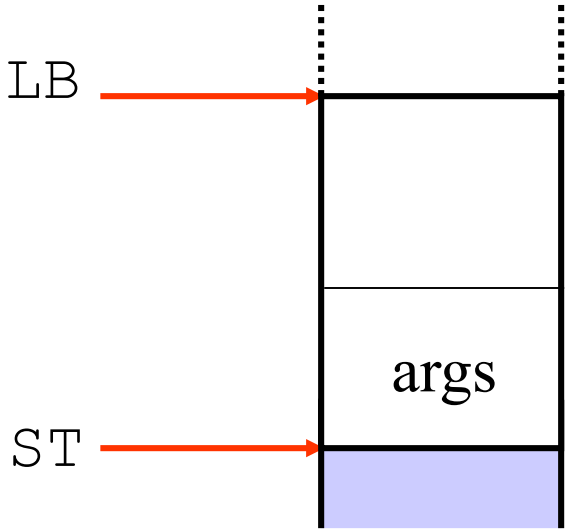
Routine Protocol



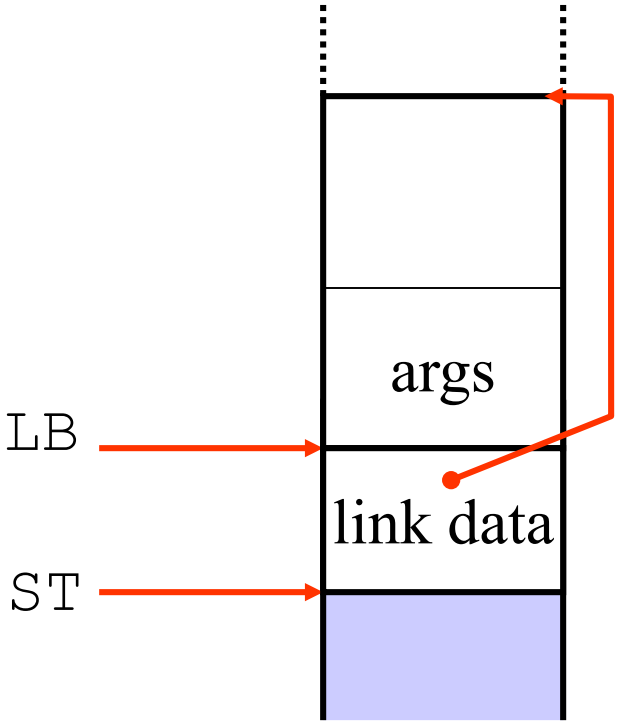
What happens in between?

Routine Protocol

(1) just before the call



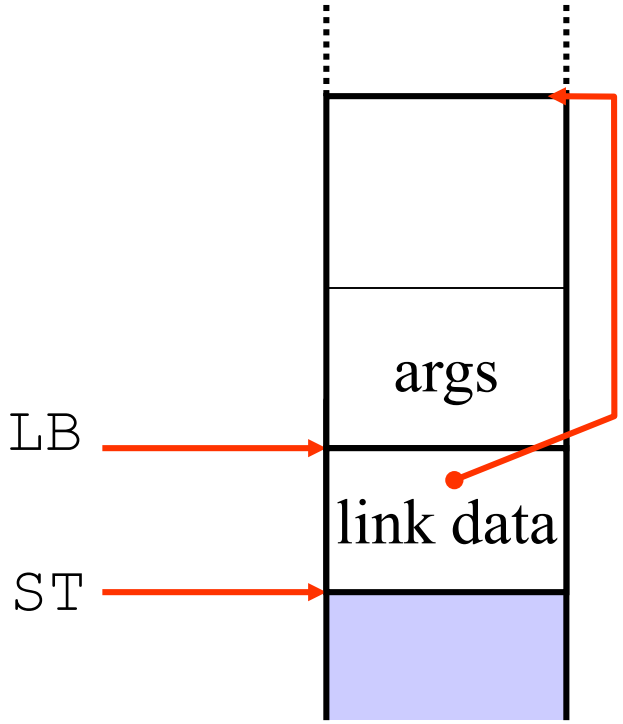
(2) just after entry



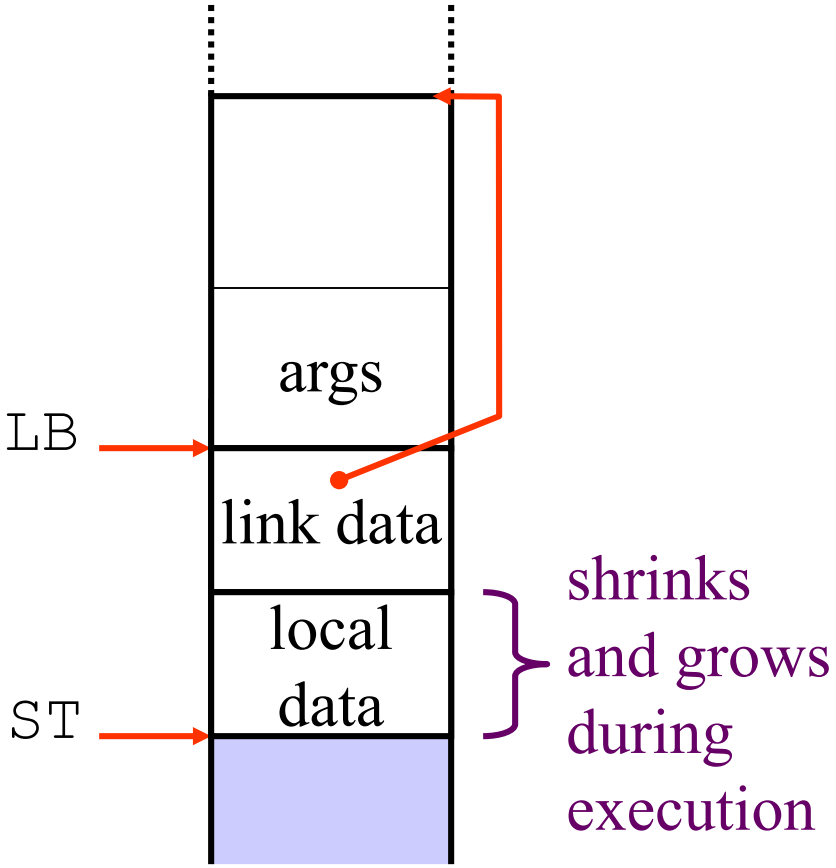
note: Going from (1) -> (2) in JVM is the execution of a single `CALL` instruction.

Routine Protocol

(2) just after entry

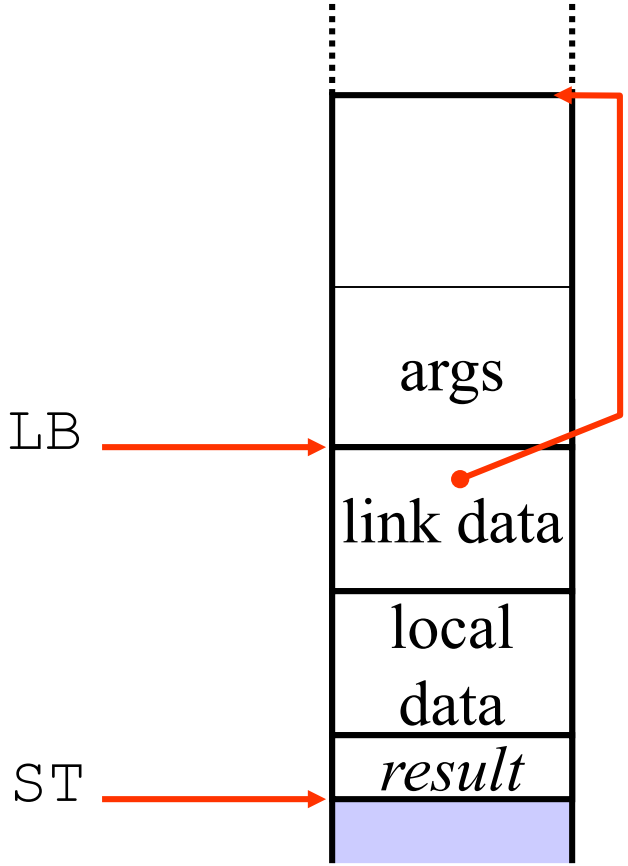


(3.1) during execution of routine

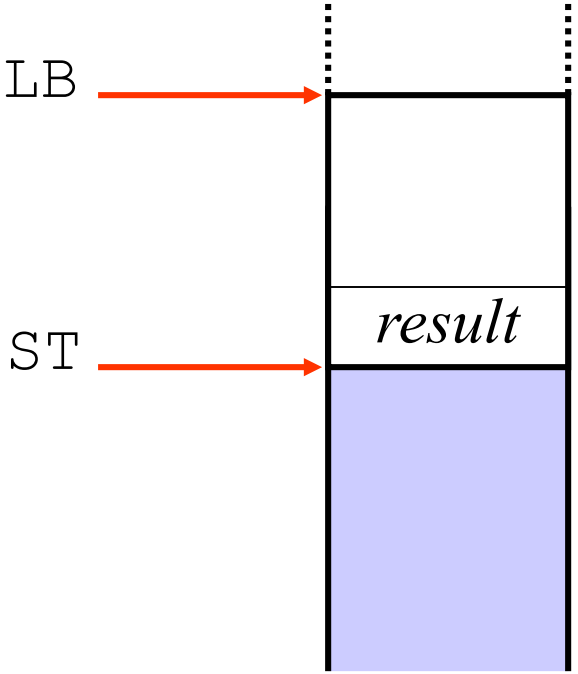


Routine Protocol

(3.2) just before return



(4) just after return



note: Going from (3.2) -> (4) in JVM is the execution of a single `RETURN` instruction.

Procedures and Functions: Parameters

We extend Mini Triangle with ...

```
Declaration
   ::= ...
     | proc Identifier (Formal) : TypeDenoter ~
                 Expression
Expression
   ::= ...
     | Identifier (Actual)
Formal
   ::= Identifier : TypeDenoter
     | var Identifier : TypeDenoter
Actual
   ::= Expression
     | var VName
```

Code Templates Parameters

elaborate [**proc** I(FP) ~ C] =

```
        JUMP  g
    e:  execute [C]
        RETURN(0)  d
    g:
```

where *d* is the size of FP

execute [I (AP)] =

```
        passArgument [AP]
        CALL(r)  e
```

Where (l,e) = address of routine bound to I,
Cl = current routine level
r = display-register(cl,l)

passArgument [E] =

```
        evaluate [E]
```

passArgument [**var** V] =

```
        fetchAddress [V]
```

Arguments: by value or by reference

Value parameters:
At the call site the argument is an expression, the evaluation of that expression leaves some value on the stack. The value is passed to the procedure/function.
A typical instruction for putting a value parameter on the stack:
`LOADL 6`

Var parameters:
Instead of passing a value on the stack, the address of a memory location is pushed. This implies a restriction that only "variable-like" things can be passed to a var parameter. In Triangle there is an explicit keyword **var** at the call-site, to signal passing a var parameter. In Pascal and C++ the reference is created implicitly (but the same restrictions apply).
Typical instructions: `LOADA 5[LB]` `LOADA 10[SB]`

Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- Production compilers do different things
  - Emphasis is on keeping values (esp. current stack frame) in registers
  - Intermediate results are laid out in the AR, not pushed and popped from the stack

Pause

Code generation for the MIPS Architecture

MIPS is implementation of a RISC architecture

- MIPS32 ISA
  - Designed for use with high-level programming languages
    - small set of instructions and addressing modes, easy for compilers
      - fixed instruction width (32-bits),
      - minimize control complexity, allow for more registers
      - 32 general purpose registers (32 bits each)
  - Arithmetic operations use registers for operands and results
    - Must use load and store instructions to use operands and results in memory
  - Load-store machine
    - large register set (32 word sized regs)
    - minimize main memory access
- MIPS has a nice simulator called SPIM
- MIPS (sometimes called RISC-I) is inspiration for the RISC-V processor

MIPS organization

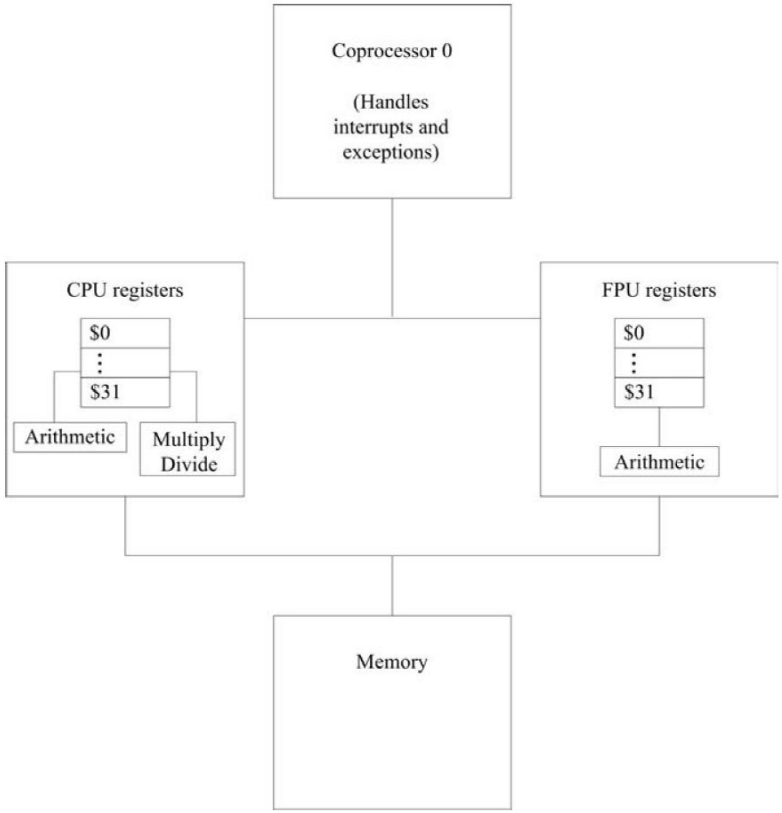


FIGURE 6.2 The MIPS computer organization.

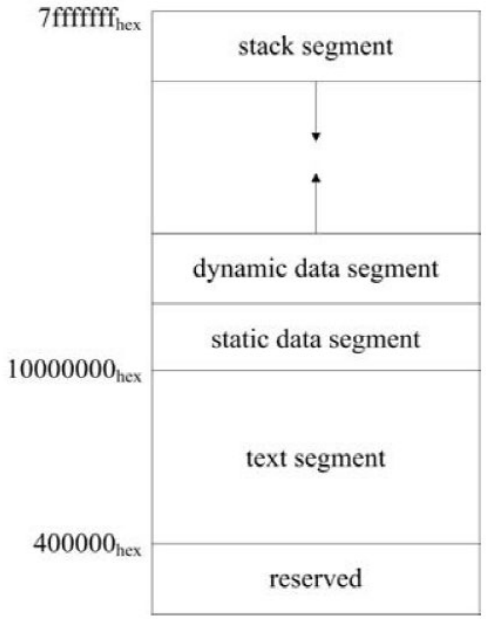


FIGURE 6.3 SPIM memory organization.

Source: Introduction to Compiler Construction in a Java World: B. Campbell et. Al.

Register conventions

register conventions and mnemonics

Number	Name	Use
0	`$zero`	hardwired 0 value
1	`$at`	used by assembler (pseudo-instructions)
2-3	`$v0-1`	subroutine return value
4-7	`$a0-3`	arguments: subroutine parameter value
8-15	`$t0-7`	temp: can be used by subroutine without saving
16-23	`$s0-7`	saved: must be saved and restored by subroutine
24-25	`$t8-9`	temp
26-27	`$k0-1`	kernel: interrupt/trap handler
28	`$gp`	global pointer (static or extern variables)
29	`$sp`	stack pointer
30	`$fp`	frame pointer
31	`$ra`	return address for subroutine
	`Hi, Lo`	used in multiplication (provide 64 bits for result)

hidden registers

PC, the program counter, which stores the current address of the instruction being executed

IR, which stores the instruction being executed

MIPS Instructions

- MIPS instructions fall into 5 classes:
  - Arithmetic/logical/shift/comparison (R-type)
  - Load/store (I-type)
  - Control instructions (branch and jump) (J-type)
  - Other (exception, register movement to/from GP registers, etc.)

- Three instruction encoding formats:
  - R-type (6-bit opcode, 5-bit rs, 5-bit rt, 5-bit rd, 5-bit shamt, 6-bit function code)

31-26	25-21	20-16	15-11	10-6	5-0
opcode	*rs*	*rt*	*rd*	shamt	function

  - I-type (6-bit opcode, 5-bit rs, 5-bit rt, 16-bit immediate)

31-26	25-21	20-16	15-0
opcode	*rs*	*rt*	*imm*

  - J-type (6-bit opcode, 26-bit pseudo-direct address)

31-26	25-0
opcode	*pseudodirect jump address*

A Sample of MIPS Instructions

- lw reg_1 offset(reg_2)
  - Load 32-bit word from address reg_2 + offset into reg_1
- add reg_1 reg_2 reg_3
  - $reg_1 \leftarrow reg_2 + reg_3$
- sw reg_1 offset(reg_2)
  - Store 32-bit word in reg_1 at address reg_2 + offset
- addiu reg_1 reg_2 imm
  - $reg_1 \leftarrow reg_2 + imm$
  - "u" means overflow is not checked
- li reg imm
  - $reg \leftarrow imm$

MIPS Addressing Modes

- MIPS addresses register operands using 5-bit field
  - Example: **`ADD $2, $3, $4`**
- Immediate addressing
  - Operand is help as constant (literal) in instruction word
  - Example: **`ADDI $2, $3, 64`**

- MIPS addresses load/store locations
  - base register + 16-bit signed offset (byte addressed)
    - Example: **`LW $2, 128($3)`**
  - 16-bit direct address (base register is 0)
    - Example: **`LW $2, 4092($0)`**
  - indirect (offset is 0)
    - Example: **`LW $2, 0($4)`**

MIPS Addressing Modes

- MIPS addresses jump targets as register content or 26-bit “pseudo-direct” address
- Example: JR $31, J 128

- MIPS addresses branch targets as signed instruction offset
  - relative to next instruction (“PC relative”)
  - in units of instructions (words)
  - held in 16-bit offset in I-type
  - Example: **`BEQ $2, $3, 12`**

A small language example

- A language with integers and integer operations

$$P \rightarrow D; P \mid D$$

$$D \rightarrow \text{def id(ARGS)} = E;$$

$$\text{ARGS} \rightarrow \text{id, ARGS} \mid \text{id}$$

$$E \rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$$

$$\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)$$

- The first function definition f is the "main" routine
- Running the program on input i means computing f(i)

Code Generation Strategy

- For each expression e we generate MIPS code that:
  - Computes the value of e in $a0
  - Preserves $sp and the contents of the stack

- We define a code generation function cgen[e] whose result is the code generated for e

Code Generation for Sub and Constants

- The code to evaluate a constant simply copies it into the accumulator:

- cgen[i] = li $a0 i

- Note that this also preserves the stack, as required

Code Generation for Add and SUB

cgen[e1 + e2] =

```
cgen[e1]
sw $a0 0($sp)
addiu $sp $sp -4
cgen[e2]
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

Cgen[e_1 - e_2] =

```
cgen[e1]
sw $a0 0($sp)
addiu $sp $sp -4
cgen[e2]
lw $t1 4($sp)
sub $a0 $t1 $a0
addiu $sp $sp 4
```

Code Generation for Conditional

- We need flow control instructions

- Instruction: beq reg_1 reg_2 label
  - Branch to label if $reg_1 = reg_2$

- Instruction: b label
  - Unconditional jump to label

Code Generation for Conditional

Cgen[if $e_1 = e_2$ then e_3 else e_4] =

```
  cgen[e1]
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen[e2]
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true_branch
  false_branch:
    cgen[e4]
    b end_if
  true_branch:
    cgen[e3]
  end_if:
```

The Activation Record

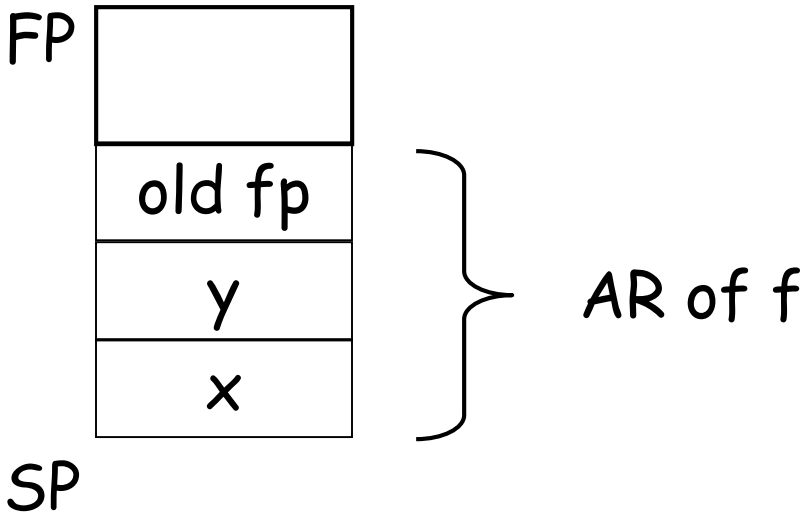
- Code for function calls and function definitions depends on the layout of the activation record
- A very simple AR suffices for this language:
  - The result is always in the accumulator
    - No need to store the result in the AR
  - The activation record holds actual parameters
    - For $f(x_1,\ldots,x_n)$ push $x_n,\ldots,x_1$ on the stack
    - These are the only variables in this language

The Activation Record (Cont.)

- The stack discipline guarantees that on function exit $sp is the same as it was on function entry
  - No need for a control link/static link
- We need the return address
- It's handy to have a pointer to the current activation
  - This pointer lives in register $fp (frame pointer)
  - Reason for frame pointer will be clear shortly

The Activation Record

- For this language, an AR with the caller's frame pointer (dynamic link), the actual parameters, and the return address suffices
- Picture: Consider a call to f(x,y), The AR will be:



Code Generation for Function Call

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation
- New instruction: jal label
  - Jump to label, save address of next instruction in $ra
  - On other architectures the return address is stored on the stack by the “call” instruction

Code Generation for Function Call (Cont.)

Cgen[f(e_1,…,e_n)] =

```
sw $fp 0($sp)
addiu $sp $sp -4
cgen[e_n]
sw $a0 0($sp)
addiu $sp $sp -4
…
cgen[e_1]
sw $a0 0($sp)
addiu $sp $sp -4
jal f_entry
```

- The caller saves its value of the frame pointer
- Then it saves the actual parameters in reverse order
- The caller saves the return address in register $ra
- The AR so far is 4*n+4 bytes long

Code Generation for Function Definition

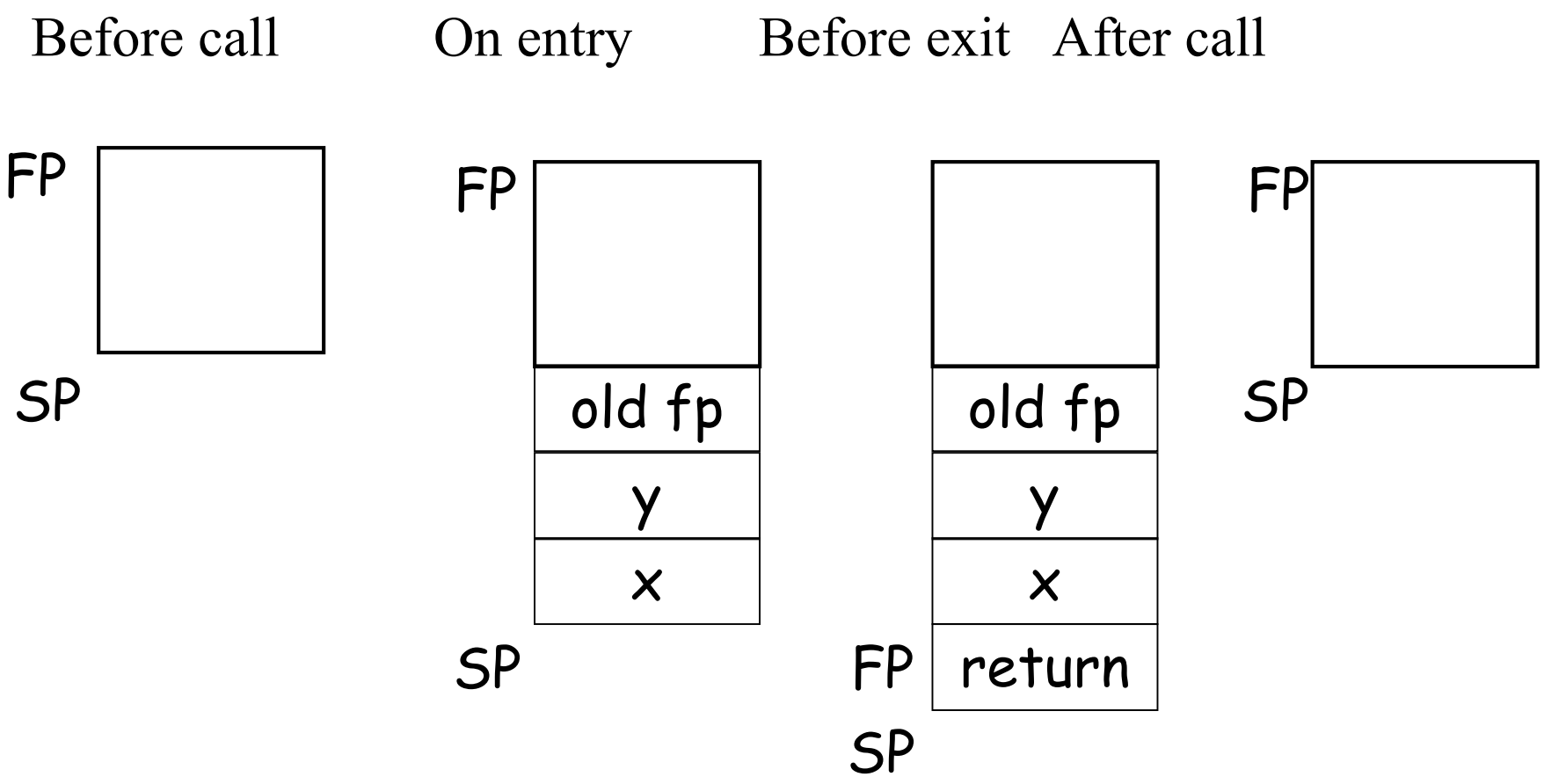
- Instruction: jr reg
  - Jump to address in register reg

Cgen[def f(x_1,…,x_n) = e] =

```
move $fp $sp
sw $ra 0($sp)
addiu $sp $sp -4
cgen[e]
lw $ra 4($sp)
addiu $sp $sp z
lw $fp 0($sp)
jr $ra
```

- Note: The frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- z = 4*n + 8

Calling Sequence. Example for f(x,y).



Code Generation for Variables

- Variable references are the last construct
- The "variables" of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller

- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp

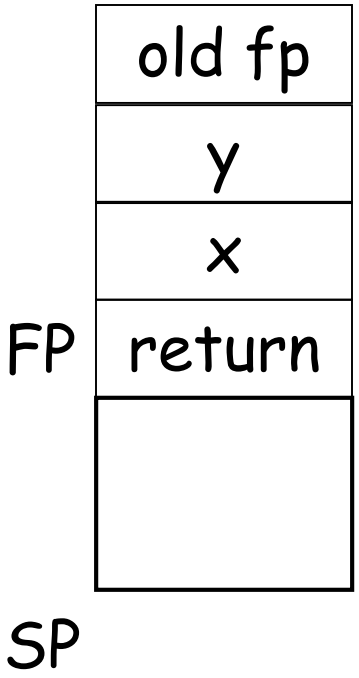
Code Generation for Variables (Cont.)

- Solution: use a frame pointer
  - Always points to the return address on the stack
  - Since it does not move it can be used to find the variables
- Let x_i be the i^{th} (i = 1,…,n) formal parameter of the function for which code is being generated

$$cgen[x_i] = \text{lw \$a0 z(\$fp)} \qquad (z = 4*i)$$

Code Generation for Variables (Cont.)

- Example: For a function def f(x,y) = e the activation and frame pointer are set up as follows:



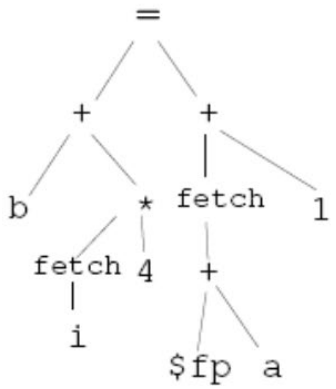
- X is at fp + 4
- Y is at fp + 8

fac(n) = if (n = 1) then 1 else (n*fac(n-1))

```
move $fp $sp                        #copy fp to top of stack
sw $ra 0($sp)                       #save ra on top of stack
addiu $sp $sp -4                                    #adjust tos
        lw $a0 4($fp)               #/load n
          sw $a0 0($sp)             # save n on tos
          addiu $sp $sp -4
          li $a0 1                  #load 1
          lw $t1 4($sp)             #load n into t1
          addiu $sp $sp 4
          beq $a0 $t1 true_branch   #branch if 1 = n
        false_branch:
                  lw $a0 4($fp)     #load n
                 sw $a0 0($sp)
                 addiu $sp $sp -4
                  lw $a0 4($fp)     #load n
                    sw $a0 0($sp)
                    addiu $sp $sp -4
                    li $a0 1        #load 1
                    lw $t1 4($sp)
                    sub $a0 $t1 $a0                 #n-1
                    addiu $sp $sp 4
                   sw $a0 0($sp)
                   addiu $sp $sp -4
                   jal f_entry      #call fac
            lw $t1 4($sp)
            mul $a0 $t1 $a0         #n*fac(n-1)
            addiu $sp $sp 4
          b end_if
          true_branch:
           li $a0 1                 #load 1
          end_if:
lw $ra 4($sp)
addiu $sp $sp 4                     #remove n from toc
lw $fp 0($sp)
jr $ra                              #return from fac
```

Pause

Instruction selection by patternmatching



Translate AST to tree rep.
with leaves corresponding
to registers, memeory locations or litterals
and internal nodes to fetch
and basic operations

Figure 13.26: Low-Level IR Representation of `b[i]=a+1`

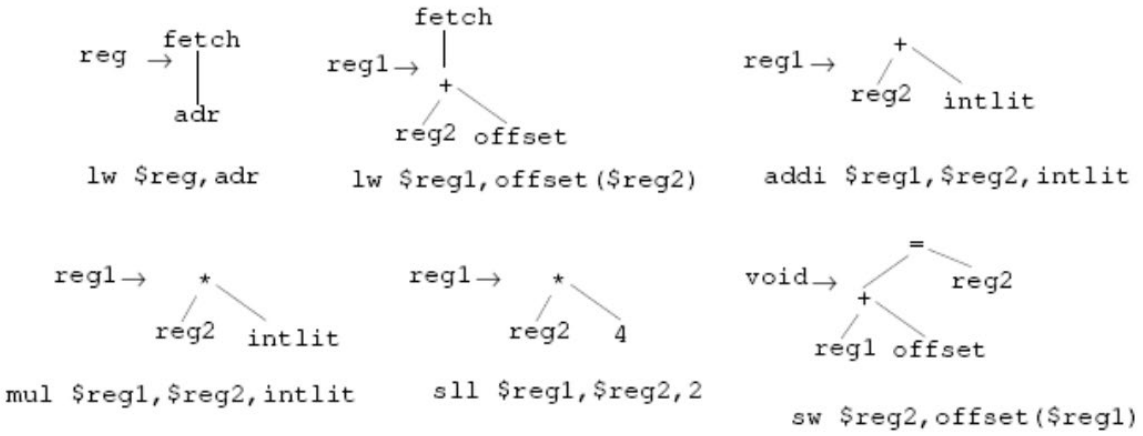


Figure 13.27: IR Tree Patterns for Various MIPS Instructions

Instruction selection
is now a question of
pattern matching
similar to bottom up
parsing

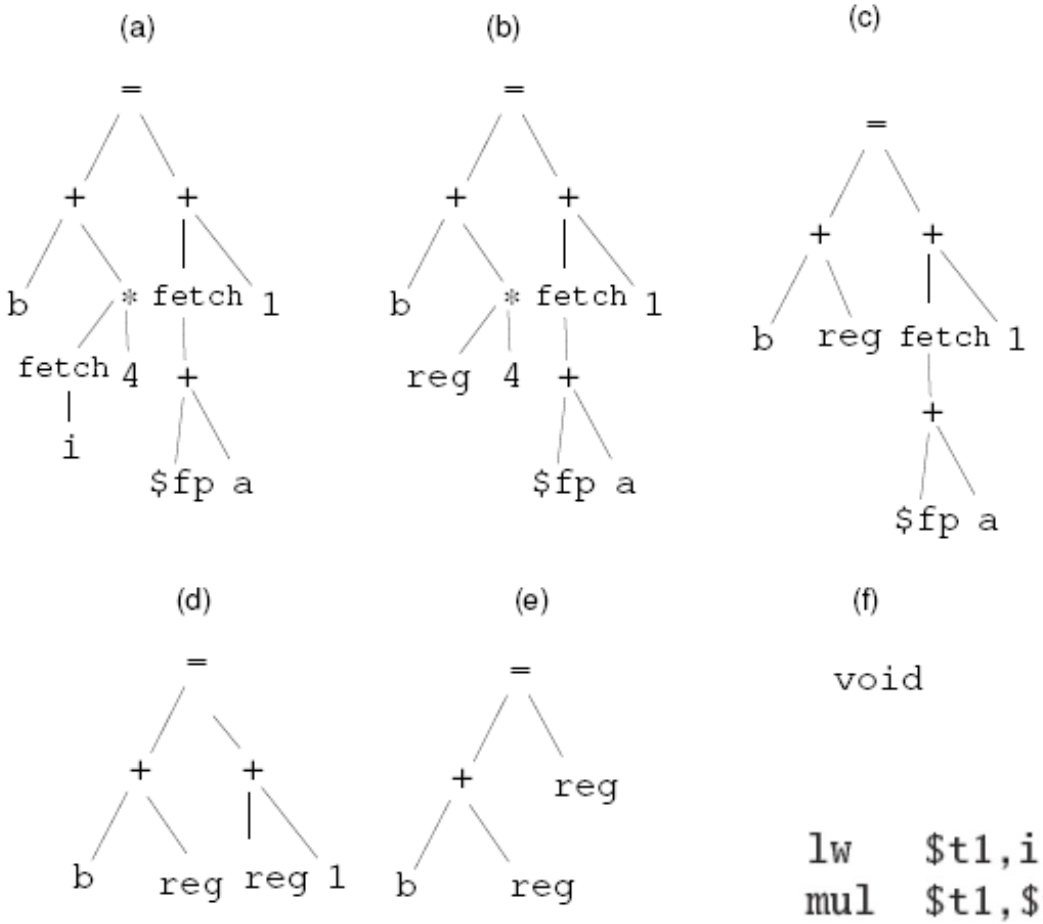
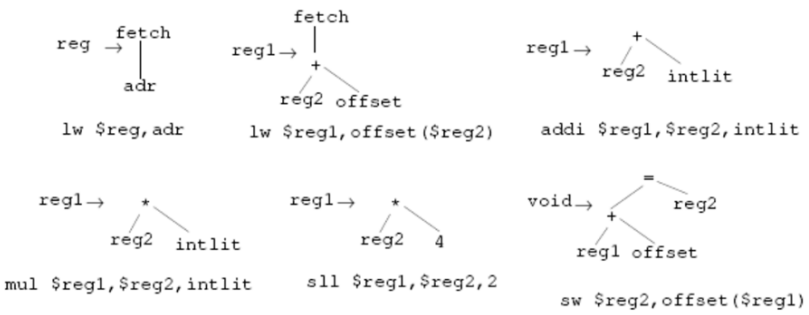


Figure 13.28: Instruction Selection Using Patterns

```
lw   $t1,i
mul  $t1,$t1,4
lw   $t2,a($fp)
addi $t2,$t2,1
sw   $t2,b($t1)
```

Figure 13.29: MIPS code for `b[i]=a+1`



Code generation from IR

- JBC to Machine code is used by AOT (Ahead-of-Time) Java compilers like gcj and FijiVM
- JBC to Machine code is used by all JIT VMs
  - Some JIT VM compile JBC on class loading
  - Others start interpretation and then compile HOT methods and store the compiled code in a method cashe
  - Others record sequences of JBC and discover ”often used sequences” and then compiles these – so called trace based JIT (e.g. Mozilla’s TraceMonkey)
- We look at JBC to MIPS

```
iload 2       ; Push int b onto stack
iload 3       ; Push int c onto stack
iadd          ; Add top two stack values
iload 4       ; Push int d onto stack
isub          ; Subtract top two stack values
istore 1      ; Store top stack value into a
```

Figure 13.1: Bytecodes for `a = b + c - d;`

```
lw      $t0,16($fp)       # Load b, at 16+$fp, into $t0
lw      $t1,20($fp)       # Load c, at 20+$fp, into $t1
add     $t2,$t0,$t1       # Add $t0 and $t1 into $t2
lw      $t3,24($fp)       # Load d, at 24+$fp, into $t3
sub     $t4,$t2,$t3       # Subtract $t3 from $t2 into $t4
sw      $t4,12($fp)       # Store result into a, at 12+$fp
```

Figure 13.2: MIPS code for `a = b + c - d;`

```
bltz    $index,badIndex       # Branch to badIndex if $index<0
lw      $temp,SIZE($array)    # Load size of array into $temp
slt     $temp,$index,$temp    # $temp = $index < size of array
beqz    $temp,badIndex        # Branch to badIndex if
                              # $index >= size of array
sll     $temp,$index,2        # multiply $index by 4 (size of
                              # an int) using a left shift
add     $temp,$temp,$array    # Compute $array + 4*$index
lw      $val,OFFSET($temp)    # Load word at
                              # $array + 4*$index + OFFSET
```

Figure 13.3: MIPS code for `iaload` bytecode

```
bltz    $index,badIndex       # Branch to badIndex if $index<0
lw      $temp,SIZE($array)    # Load size of array into $temp
slt     $temp,$index,$temp    # $temp = $index < size of array
beqz    $temp,badIndex        # Branch to badIndex if
                              # $index >= size of array
sll     $temp,$index,2        # multiply $index by 4 (size of
                              # an int) using a left shift
add     $temp,$temp,$array    # Compute $array + 4*$index
sw      $val,OFFSET($temp)    # Load $val into word at
                              #  $array + 4*$index + OFFSET
```

Figure 13.4: MIPS code for `iastore` bytecode

```
move    $a0,$t0              # Copy $t0 to parm register 1
li      $a1,2                # Load 2 into parm register 2
sw      $t0,32($fp)          # Store $t0 across call
jal     f                    # Call function f
                             # Function value is in $v0
lw      $t0,32($fp)          # Restore $t0
sw      $v0,a                # Store function value in a
```

Figure 13.5: MIPS code for the function call `a = f(i,2);`

```
subi    $sp,$sp,frameSz      # Push frame on stack
sw      $ra,0($sp)           # Save return address in frame
sw      $fp,4($sp)           # Save old frame pointer in frame
move    $fp,$sp              # Set $fp to access new frame
# Save callee-save registers (if any) here
# Body of method is here
# Restore callee-save registers (if any) here
lw      $ra,0($fp)           # Reload return address register
lw      $fp,4($fp)           # Reload old frame pointer
addi    $sp,$sp,frameSz      # Pop frame from stack
jr      $ra                  # Jump to return address
```

Figure 13.6: MIPS prologue and epilogue code

Stringsum example

```
public static String stringSum(int limit){
        int sum = 0;
        for (int i = 1; i <= limit; i++)
                sum += i;
        return Integer.toSting(sum);
}
```

```
    iconst_0      ; Push 0
    istore_1      ; Store into variable #1 (sum)
    iconst_1      ; Push 1
    istore_2      ; Store into variable #2 (i)
    goto L2       ; Go to end of loop test
L1: iload_1       ; Push var #1 (sum) onto stack
    iload_2       ; Push var #2 (i) onto stack
    iadd          ; Add sum + i
    istore_1      ; Store sum + i into var #1 (sum)
    iinc 2 1      ; Increment var #2 (i) by 1
L2: iload_2       ; Push var #2 (i)
    iload_0       ; Push var #0 (limit)
    if_icmple L1  ; Goto L1 if i <= limit
    iload_1       ; Push var #1 (sum) onto stack
                  ; Call toString:
    invokestatic
          java/lang/Integer/toString(I)Ljava/lang/String;
    areturn       ; Return String reference to caller
```

Figure 13.7: Bytecodes for method `stringSum`

```
        subi    $sp,$sp,20      # Push frame on stack
        sw      $ra,0($sp)      # Save return address
        sw      $fp,4($sp)      # Save old frame pointer
        move    $fp,$sp         # Set $fp to access new frame
        sw      $a0,8($fp)      # Store limit in frame
        sw      $0,12($fp)      # Store 0 ($0) into sum
        li      $t0,1           # Load 1 into $t0
        sw      $t0,16($fp)     # Store 1 into i
        j       L2              # Go to end of loop test
L1:     lw      $t1,12($fp)     # Load sum into $t1
        lw      $t2,16($fp)     # Load i into $t2
        add     $t3,$t1,$t2     # Add sum + i into $t3
        sw      $t3,12($fp)     # Store sum + i into sum
        lw      $t4,16($fp)     # Load i into $t2
        addi    $t4,$t4,1       # Increment $t4 by 1
        sw      $t4,16($fp)     # Store $t4 into i
L2:     lw      $t5,16($fp)     # Load i into $t5
        lw      $t6,8($fp)      # Load limit into $t6
        sle     $t7,$t5,$t6     # set $t7 = i <= limit
        bnez    $t7,L1          # Goto L1 if i <= limit
        lw      $t8,12($fp)     # Load sum into $t8
        move    $a0,$t8         # Copy $t8 to parm register
        jal     String_toString_int_   # Call toString
                                # String ref now is in $v0
        lw      $ra,0($fp)      # Reload return address
        lw      $fp,4($fp)      # Reload old frame pointer
        addi    $sp,$sp,20      # Pop frame from stack
        jr      $ra             # Jump to return address
```

Figure 13.8: MIPS code for method `stringSum`

Register Allocation

- A compiler generating code for a register machine needs to pay attentention to register allocation as this is a limited ressource
- In routine protocol
  - Allocate arg1 in R1, arg2 in R2 .. Result in R0
  - But what if there are more args than regs?
- In evaluation of expressions
  - On MIPS all calculations take place in regs
  - Reduce traffic between memory and regs

```
procedure registerNeeds(T)
    if T.kind = Identifier or T.kind = IntegerLiteral
    then T.regCount ← 1
    else
        call registerNeeds(T.leftChild)
        call registerNeeds(T.rightChild)
        if T.leftChild.regCount = T.rightChild.regCount
        then T.regCount ← T.rightChild.regCount + 1
        else
            T.regCount ← max(T.leftChild.regCount, T.rightChild.regCount)
end
```

Figure 13.9: An Algorithm to Label Expression Trees with Register Needs

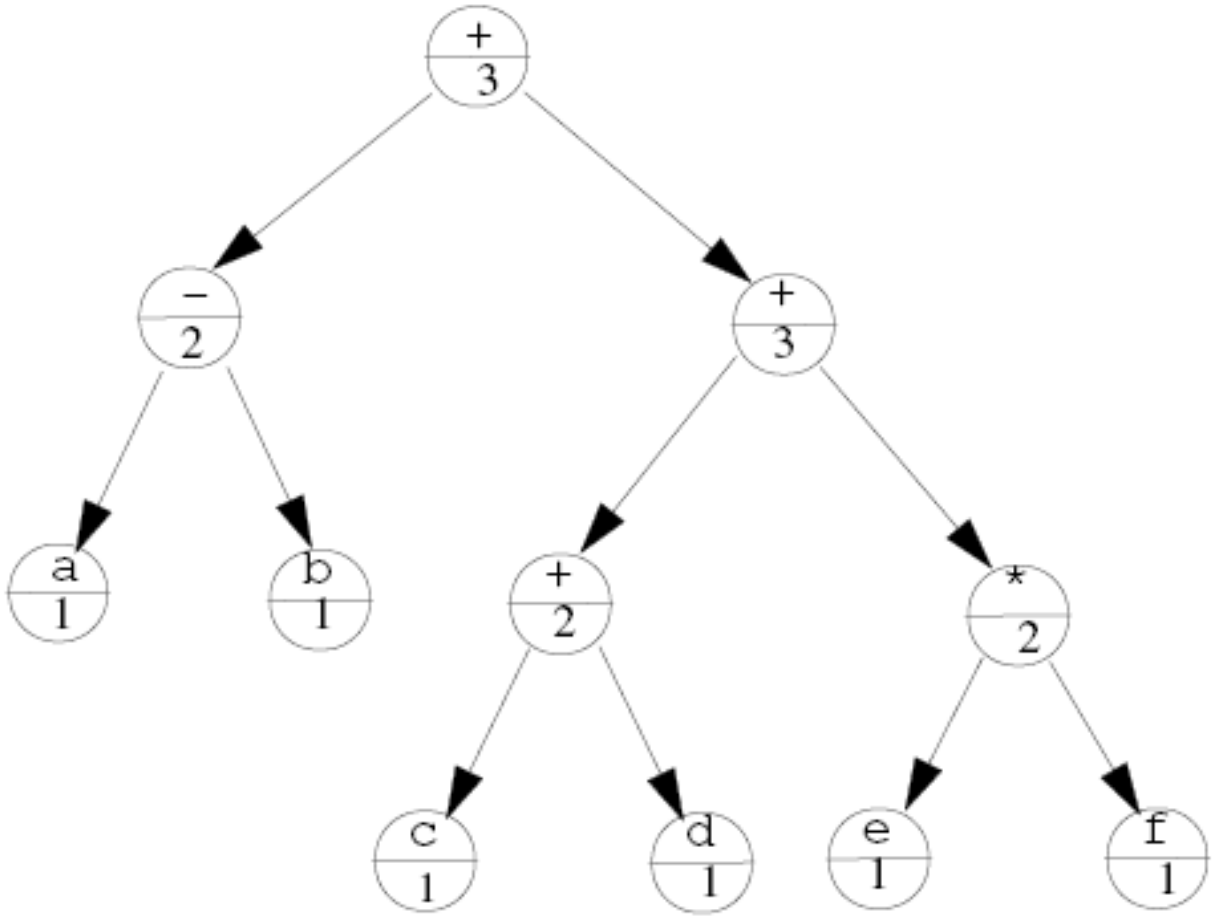


Figure 13.10: Expression Tree for `(a-b) + ((c+d)+(e*f))` with Register Needs.

```
lw   $10, c          # Load c into register 10
lw   $11, d          # Load d into register 11
add  $10, $10, $11   # Compute c + d into register 10
lw   $11, e          # Load e into register 11
lw   $12, f          # Load f into register 12
mul  $11, $11, $12   # Compute e * f into register 11
add  $10, $10, $11   # Compute (c + d) + (e * f) into reg 10
lw   $11, a          # Load a into register 11
lw   $12, b          # Load b into register 12
sub  $11, $11, $12   # Compute a - b into register 11
add  $10, $11, $10   # Compute (a-b)+((c+d)+(e*f)) into reg 10
```

Figure 13.11: MIPS code for `(a-b) + ((c+d)+(e*f))`

```
procedure TREECG(T, regList)
    r1 ← HEAD(regList)
    r2 ← HEAD(TAIL(regList))
    if T.kind = Identifier
    then
        /* Load a variable. */
        call GENERATE(lw, r1, T.IdentifierName)
    else
        if T.kind = IntegerLiteral
        then
            /* Load a literal. */
            call GENERATE(li, r1, T.IntegerValue)
        else
            /* T.kind must be a binary operator. */
            left ← T.leftChild
            right ← T.rightChild
            if left.regCount ≥ LENGTH(regList) and right.regCount ≥ LENGTH(regList)
            then
                /* Must spill a register into memory. */
                call TREECG(left, regList)
                /* Get memory location. */
                temp ← GETTEMP( )
                call GENERATE(sw, r1, temp)
                call TREECG(right, regList)
                call GENERATE(lw, r2, temp)
                /* Free memory location. */
                call FREETEMP(temp)
                call GENERATE(T.operation, r1, r2, r1)
            else
                /* There are enough registers; no spilling is needed. */
                if left.regCount ≥ right.regCount
                then
                    call TREECG(left, regList)
                    call TREECG(right, TAIL(regList))
                    call GENERATE(T.operation, r1, r1, r2)
                else
                    call TREECG(right, regList)
                    call TREECG(left, TAIL(regList))
                    call GENERATE(T.operation, r1, r2, r1)
end
```

Figure 13.12: An Algorithm to Generate Optimal Code from Expression Trees

Optimizing register allocations

- TreeCG generates code such that result(s) end up in targeted registers
- However TreeCG does not exploit communicative operators
  - exp1 op exp2 = exp2 op exp1
  - Also difficult due to overflow or exceptions
- Exploiting associativity can reduce reg needs
  - (a+b)+(c+d) needs 3 regs
  - a+b+c+d needs only 2 regs

Register Allocation

- Expression level register allocation
- Procedure level register allocation
  - Interference graphs
  - Graph coloring
- Intra-procedural register allocation
  - 10%-28% speed-up

Code scheduling

- Modern computers are pipelined
  - Instructions are processed in stages
  - Instructions take different time to execute
  - If result from previous instruction is needed but not yet ready then we have a **stalled pipeline**
  - Delayed load
    - Load from memory takes 2, 10 or 100 cycles
  - Also FP instructions takes time

```
1. lw  $10,a            6.  add  $10,$10,$12
2. lw  $11,b            7.  mul  $11,$11,$10
3. mul $11,$10,$11      8.  mul  $12,$10,$12
4. lw  $10,c            9.  add  $12,$11,$12
5. lw  $12,d            10. sw   $12,a
```

Figure 13.21: MIPS code for `a=((a*b)*(c+d))+(d*(c+d))`

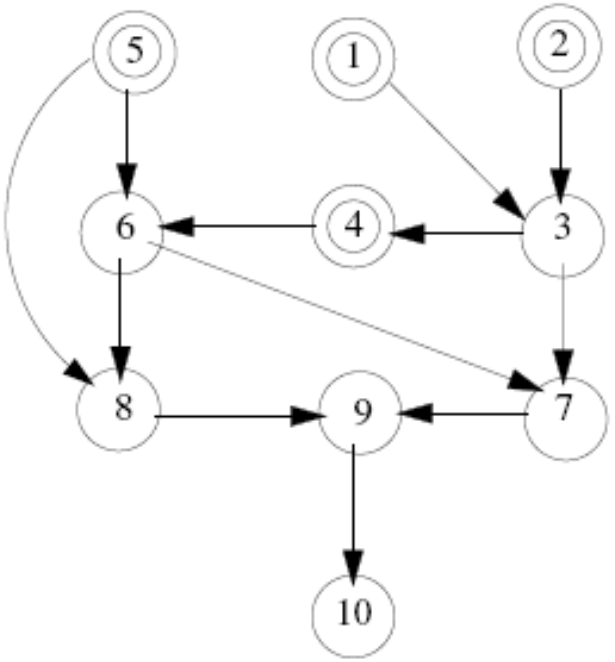


Figure 13.22: Dependency DAG for `a=((a*b)*(c+d))+(d*(c+d))`

```
procedure SCHEDULEDAG(dependencyDAG)
    candidates ← ROOTS(dependencyDAG)
    while candidates ≠ ∅ do
        call SELECT(candidates, "Is not stalled by last instruction generated")
        call SELECT(candidates, "Can stall some successor")
        call SELECT(candidates, "Exposes the most new roots if generated")
        call SELECT(candidates, "Has the longest path to a leaf")
        inst ← Any node ∈ candidates
        Schedule inst as next instruction to be executed
        dependencyDAG ← dependencyDAG − {inst}
        candidates ← ROOTS(dependencyDAG)
end
```

Figure 13.23: An Algorithm to Schedule Code from a Dependency DAG

```
1. lw  $10,a          6. add $10,$10,$12
2. lw  $11,b          7. mul $11,$11,$10
3. lw  $12,d          8. mul $12,$10,$12
4. mul $11,$10,$11    9. add $12,$11,$12
5. lw  $10,c          10 sw  $12,a
```

Figure 13.24: Scheduled MIPS code for `a=((a*b)*(c+d))+(d*(c+d))`

Reg allocation and Code Scheluling

- Reg allocations algorithms try to minimize the number of regs used
- May conflict with pipeline architecture
  - Using more regs than strictly necessary may avoid pipeline stalls
- Solution
  - Integrated register allocator and code scheduler

```
1. lw  $10,a          6.  add  $10,$13,$12
2. lw  $11,b          7.  mul  $11,$11,$10
3. lw  $12,d          8.  mul  $12,$10,$12
4. lw  $13,c          9.  add  $12,$11,$12
5. mul $11,$10,$11    10. sw   $12,a
```

Figure 13.25: Delay-free MIPS code for `a=((a*b)*(c+d))+(d*(c+d))`

Modern Hardware and code generation

- Speculative execution
- Prefetch instructions
  - Load data into cache
- Dynamic scheduling
- Out of order architectures

- Should the HW, Compiler or the programmer do the job?

Register variable in C

- Ex: `register float a = 0 ;`

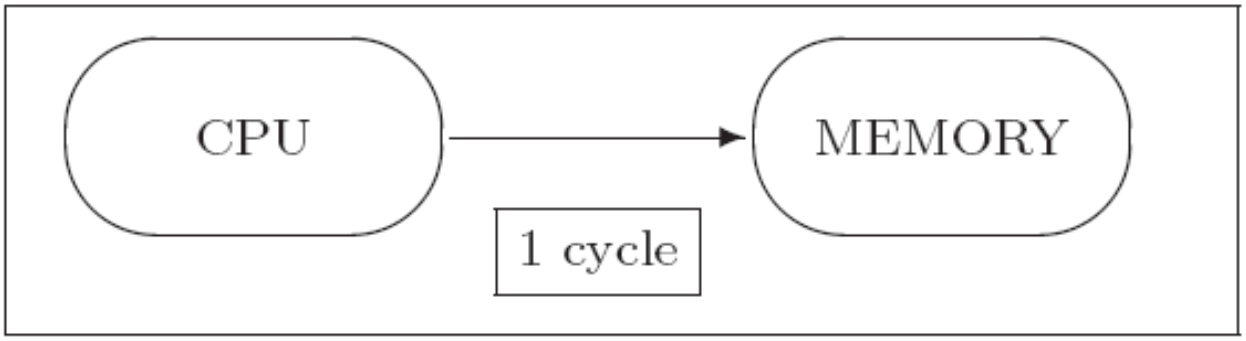
- register provides a hint to the compiler that you think a variable will be frequently used
- compiler is free to ignore register hint
- if ignored, the variable is equivalent to an auto variable with the exception that you may not take the address of a register (since, if put in a register, the variable will not have an address)
- rarely used, since any modern compiler will do a better job of optimization than most programmers

Java Memory Model

- Abstract memory model
  - Local stack for each thread
    - But stacks may need to be implemented via registers and memory
  - Shared variables can be problematic on some implementations
    - Serial to concurrent
      - Code for serial execution may not work in concurrent system
    - Concurrent to serial
      - Code with synchronization may be inefficient in serial programs (10-20% unnecessary overhead)

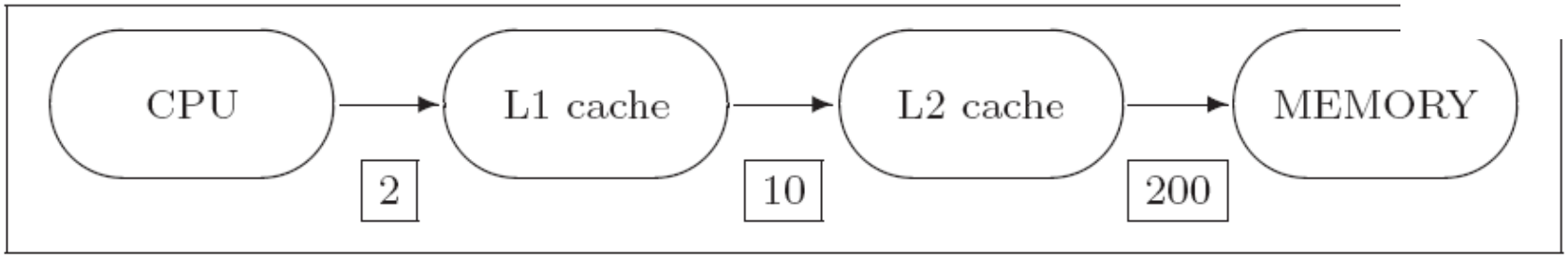
  - Java 1.5 has expanded the definition of the memory model
    - Volatile keyword
      - The value of a volatile variable will **never be cached thread-locally**: all reads and writes will go straight to "main memory"

A programmer's view of memory



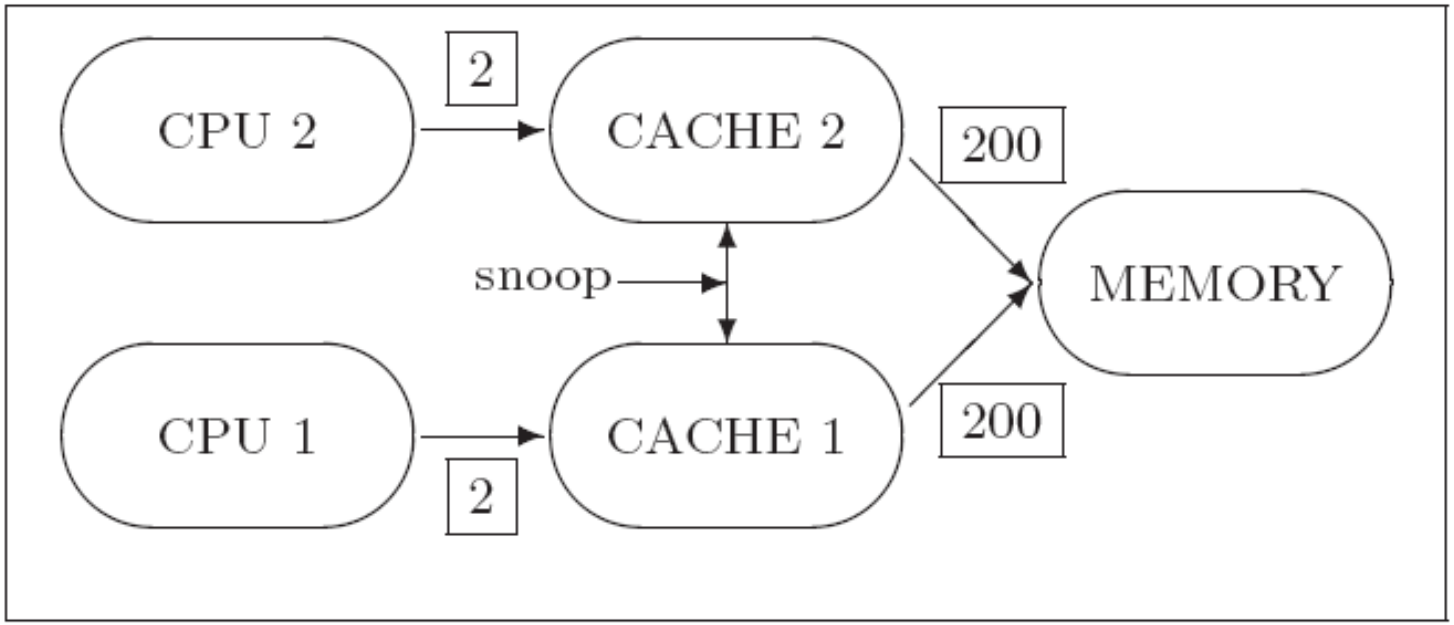
This model was pretty accurate in 1985.
Processors (386, ARM, MIPS, SPARC) all ran at 1–10MHz clock speed and could access external memory in 1 cycle; and most instructions took 1 cycle.
Indeed the C language was as expressively time-accurate as a language could be: almost all C operators took one or two cycles.
But this model is no longer accurate!

A modern view of memory timings



So what happened?
On-chip computation (clock-speed) sped up
faster (1985–2005) than off-chip communication (with memory) as feature sizes shrank.
The gap was filled by spending transistor budget on caches which (statistically) filled the mismatch until 2005 or so.
Techniques like caches, deep pipelining with bypasses, and superscalar instruction issue burned power to preserve our illusions.
2005 or so was crunch point as faster, hotter, single-CPU Pentiums were scrapped. These techniques had delayed the inevitable.

The Current Mainstream Processor



Will scale to 2, 4 maybe 8 processors.
But ultimately shared memory becomes the bottleneck (1024 processors?!?).

Conclusions

- Low level code genrations requires attentions to lots of details:
  - Instruction sequence selection
  - Register allocation
  - Instruction scheduling
  - Storage allocation
    - Memory hierachies
  - (multi-core placement)
- Sometimes Implications for language design
  - E.g. high level memory models

What can you do in your projects now?

- You should by now have lexer, parser and AST in place
  - Write pretty printer to test front end
    - Use all the programs you wrote when designing your syntax
- You should have static semantic analyzer in place.
  - Write recursive interpreter to test programs
  - And generate ideas for formal semantics
- Code generation:
  - Write C, Java, python … code generator
  - Write JBC or CIL code generator
  - Write MIPS, AVR or x86