

COMP SCI 4TB3 / 6TB3 Compiler Construction

Emil Sekerinski
Department of Computing and Software
McMaster University

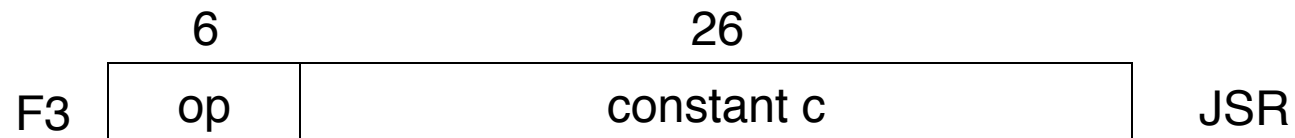
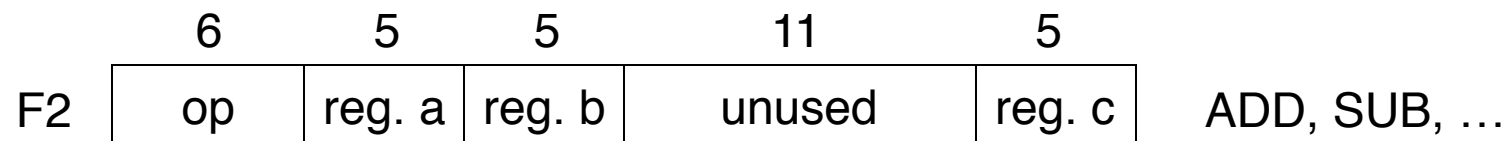
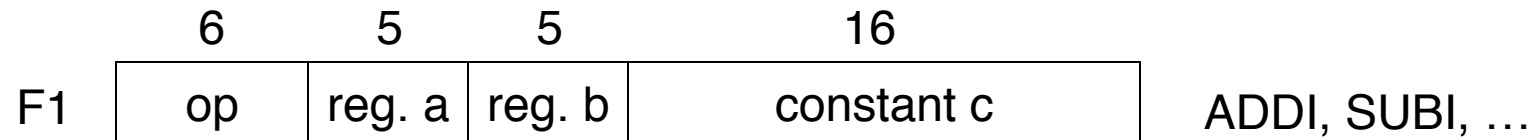
9. A RISC Architecture as Target

- The architecture described here is a real machine, in contrast to a virtual machine. It follows closely the now predominant line of RISC (Reduced Instruction Set Computer) architectures:
 - few number of instructions compared to CISC architectures
 - all instructions of same length
 - large number of registers
 - only register based operations
- The machine considered here is the commercial MIPS architecture. It is similar to the DLX architecture in the book of Hennessy & Patterson and.

Registers and Memory

- Registers R0 - R31, each 32 bits wide.
 - R0 is always zero
 - R31 used for return address for subroutine calls
- Instruction register IR, holding current instruction, 32 bits wide.
- Program counter PC, holding address of next instruction, 32 bits wide, holds word address (not byte address).
- Memory with 32 bit words, byte addressed, i.e. word addresses are multiples of 4.

Instruction Formats



Register Instructions – Arithmetic

Format F2:

ADD a,b,c	R.a := R.b + R.c; PC := PC + 1
SUB a,b,c	R.a := R.b – R.c; PC := PC + 1
MUL a,b,c	R.a := R.b * R.c; PC := PC + 1
DIV a,b,c	R.a := R.b div R.c; PC := PC + 1
MOD a,b,c	R.a := R.b mod R.c; PC := PC + 1
CMP a,b,c	R.a := R.b – R.c; PC := PC + 1
CHK a,c	if $0 \leq R.a$ and $R.a \leq R.c$ then PC := PC + 1 else PC := trap

Format F1:

ADDI a,b,c	R.a := R.b + c; PC := PC + 1
SUBI a,b,c	R.a := R.b – c; PC := PC + 1
MULI a,b,c	R.a := R.b * c; PC := PC + 1
DIVI a,b,c	R.a := R.b div c; PC := PC + 1
MODI a,b,c	R.a := R.b mod c; PC := PC + 1
CMPI a,b,c	R.a := R.b – c; PC := PC + 1
CHKI a,c	if $0 \leq R.a$ and $R.a \leq c$ then PC := PC + 1 else PC := trap

All arithmetic instructions should indicate arithmetic overflow if the result is $< -2^{31}$ or $\geq 2^{31}$. In case of overflow, CMP yields a result with wrong magnitude but correct sign.

Register Instructions – Logical and Bitwise

Format F2:

AND a,b,c R.a := R.b and R.c;
 PC := PC + 1

BIC a,b,c R.a := R.b and not(R.c);
 PC := PC + 1

OR a,b,c R.a := R.b or R.c;
 PC := PC + 1

XOR a,b,c R.a := R.b xor R.c;
 PC := PC + 1

LSH a,b,c if $c \geq 0$
 then R.a := R.b lsl R.c
 else R.a := R.b lsr -R.c;
 PC := PC + 1

ASH a,b,c if $c \geq 0$
 then R.a := R.b asl R.c
 else R.a := R.b asr -R.c;
 PC := PC + 1

Format F1:

ANDI a,b,c R.a := R.b and c;
 PC := PC + 1

BICl a,b,c R.a := R.b and not(c);
 PC := PC + 1

ORl a,b,c R.a := R.b or c;
 PC := PC + 1

XORl a,b,c R.a := R.b xor c;
 PC := PC + 1

LSHl a,b,c if $c \geq 0$
 then R.a := R.b lsl c
 else R.a := R.b lsr -c;
 PC := PC + 1

ASHl a,b,c if $c \geq 0$
 then R.a := R.b asl c
 else R.a := R.b asr -c;
 PC := PC + 1

Load and Store Instructions

Format F1:

LDW a,b,c	$R.a := M[(R.b + c) \text{ div } 4]; PC := PC + 1$	load word
LDB a,b,c	...	load byte
POP a,b,c	$R.a := M[R.b \text{ div } 4]; R.b := R.b + c; PC := PC + 1$	pop stack
STW a,b,c	$M[(R.b + c) \text{ div } 4] := R.a; PC := PC + 1$	store word
STB a,b,c	...	store byte
PSH a,b,c	$R.b := R.b - c; M[R.b \text{ div } 4] := R.a; PC := PC + 1$	push stack

Control Instructions

Format F1:

BEQ a,c	if $R.a = 0$ then $PC := PC + c$ else $PC := PC + 1$
BNE a,c	if $R.a \neq 0$ then $PC := PC + c$ else $PC := PC + 1$
BLT a,c	if $R.a < 0$ then $PC := PC + c$ else $PC := PC + 1$
BGE a,c	if $R.a \geq 0$ then $PC := PC + c$ else $PC := PC + 1$
BGT a,c	if $R.a > 0$ then $PC := PC + c$ else $PC := PC + 1$
BLE a,c	if $R.a \leq 0$ then $PC := PC + c$ else $PC := PC + 1$
BSR c	$R31 := (PC + 1) * 4$; $PC := PC + c$

Format F3:

JSR c	$R31 := (PC + 1) * 4$; $PC := c$
-------	-----------------------------------

Format F2:

RET c	$PC := R.c \text{ div } 4$
-------	----------------------------

Input/Output Instructions

Format F2:

RD a read (R.a); PC := PC + 1


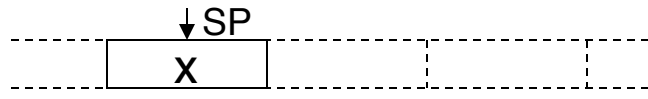
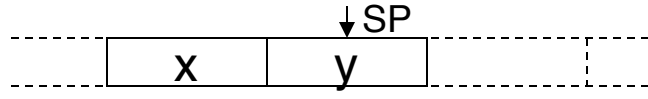
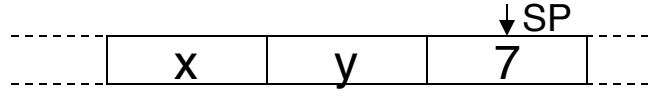
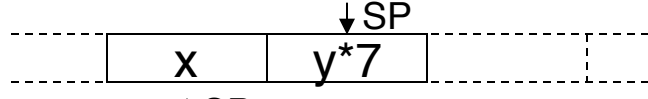
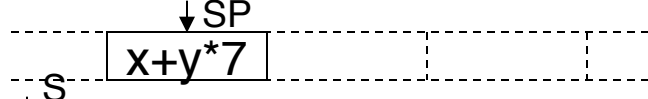
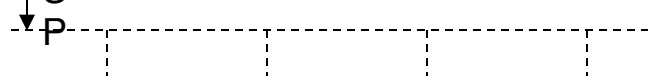
WRD c write (R.c); PC := PC + 1

WRL writeln; PC := PC + 1

These instructions are not typical computer instructions; they have been added here to provide a simple means for input and output.

Code Generation for Expressions and Assignments

- Expressions are most easily evaluated by storing the intermediate results on a stack. Stack-based architectures directly provide load, store, arithmetic, etc operations on a stack-structured memory. For register-based architectures, a stack is simulated by the available set of registers. For example, for $u := x + y * 7$:

Stack (SP = stack pointer)	Register Operations	RISC code
	$R1 := x$	LDW 1,0,adr(x)
	$R2 := y$	LDW 2,0,adr(y)
	$R3 := 7$	ADDI 3,0,7
	$R2 := R2 * R3$	MUL 2,2,3
	$R1 := R1 + R2$	ADD 1,1,2
	$u := R1$	STW 1,0,adr(u)
		

Scheme for Generating Code

K	Code(K)	Side-effect
ident	LDW SP, 0, adr(ident)	SP := SP + 1
number	ADDI SP, 0, number	SP := SP + 1
(exp)	Code(exp)	
fac ₀ * fac ₁	Code(fac ₀) Code(fac ₁) MUL SP, SP, SP + 1	SP := SP - 1
term ₀ + term ₁	Code(term ₀) Code(term ₁) ADD SP, SP, SP+1	SP := SP - 1
ident := exp	Code(exp) STW SP, 0, adr(ident)	SP := SP - 1

- A global variable SP indicates what registers are in use. The scheme does not require that R1 is the first available register. There is even no need that registers are occupied consecutively.

Improving the Generated Code

- The code generated according to this scheme is always correct, but not always optimal. For example, for $y := x + 1$ we get:

R1 := x	LDW 1,0,adr(x)
R2 := 1	ADDI 2,0,1
R1 := R1 + R2	ADD 1,1,2
y := R1	STW 1,0,adr(y)

A better code using immediate addressing mode would be:

R1 := x	LDW 1,0,adr(x)
R1 := R1 + 1	ADDI 1,1,1
y := R1	STW 1,0,adr(y)

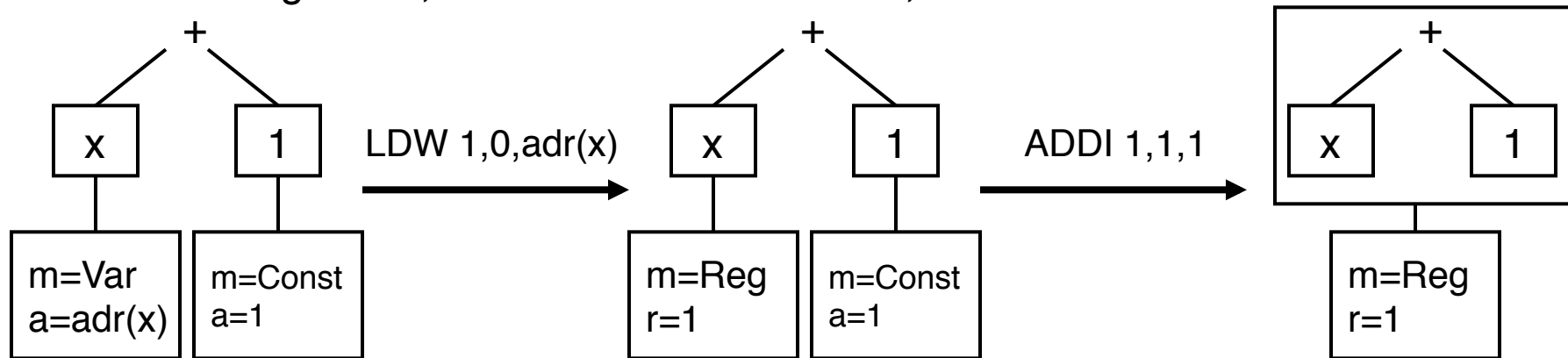
The reason for this is that the instruction for loading 1 into a register has been emitted prematurely. The remedy is therefore to delay code generation until it is definitely known that no better solution exists.

Delayed Code Generation

- The principle of delayed code generation is that all procedures for expressions (expression, SimpleExpression, term, factor), rather than generating code for storing the result on top of the stack, return a synthesized attribute with the information where the result can be found. Three possibilities exist:
 - The data is in a memory location;
 - The data is a constant;
 - The data is in a register;
 - The data is the result of a relational comparison of operands that are either in registers or constants.

Transformation of Items during Delayed Code Generation

- Parsing $x + 1$, which involves two items, one for x and one for 1 :



- Procedure factor parsing x returns in an item that x is a variable in a certain memory location. Procedure parsing 1 returns in an item that 1 is a constant with value 1 .
- Procedure SimpleExpression parsing $+$ gets both items. For an ADD instruction, the first operand has to be in a register (and the second can be either in a register or a constant). Code for loading x in a register is generated and the item for x is updated.
- Code for the addition is generated and an item that the result is in a register is returned to the calling procedure statement.

Extensions for Code Generation

- Generated code is stored in the list `asm` of triples with label, instruction, and target:
- ```
def putInstr(instr, target = ""):
 """Emit an instruction"""
 asm.append((" ", instr, target))
```
- ```
def putOp(op, a, b, c):  
    """Emit instruction op with three operands, a, b, c;  
    c can be register or immediate"""  
    putInstr(op + ' ' + a + ', ' + b + ', ' + str(c))
```
- ```
def putBranchOp(op, a, b, c):
 putInstr(op + ' ' + a + ', ' + b, str(c))
```
- ```
def putMemOp(op, a, b, c):  
    """Emit load/store instruction at location or register b + offset c"""  
    if b == R0: putInstr(op + ' ' + a + ', ' + str(c))  
    else: putInstr(op + ' ' + a + ', ' + str(c) + '(' + b + ')')
```

Code Generation for Expressions ...

- For the moment, we consider only integer expressions:

```
def factor():
    ... error handling
    if SC.sym == IDENT:
        x = find(SC.val)
        if type(x) in {Var, Ref}: x = CG.genVar(x); getSym()
        elif type(x) == Const: x = Const(x.tp, x.val); x = CG.genConst(x); getSym()
        else: mark('expression expected')
        x = selector(x)
    elif SC.sym == NUMBER:
        x = Const(Int, SC.val); x = CG.genConst(x); getSym()
    elif SC.sym == LPAREN:
        getSym(); x = expression()
        if SC.sym == RPAREN: getSym()
        else: mark("") expected")
    elif ...
    return x
```


... Code Generation for Expressions ...

- Procedure `genConst(x)` does not need to generate any code.

```
def genConst(x):  
    return x
```
- Procedure `genVar(x)` allows `x` to refer to a global variable, local variable, or procedure parameter: the assumption is that `x.reg` (which can be `R0`) and `x.adr` (which can be `0`) refer to the variable. References to variables on intermediate level is not supported. For global variables, the reference is kept symbolic, to be resolved later by the assembler. Item `x` is `Var` or `Ref`; if it is `Ref`, the reference is loaded into a new register. A new `Var` item with the location is returned.
- ```
def genVar(x): # version not supporting parameters in registers
 if 0 < x.lev < curlev: mark('level!')
 y = Var(x.tp); y.lev = x.lev
 if type(x) == Ref: # reference is loaded into register
 y.reg, y.adr = obtainReg(), 0 # variable at M[y.reg]
 putMemOp('lw', y.reg, x.reg, x.adr)
 elif type(x) == Var:
 y.reg, y.adr = x.reg, x.adr
 else: assert False
 return y
```

## ... Code Generation for Expressions ...

---

Procedure `term()` parses

`term ::= factor {"×" | "div" | "mod" | "and"} factor`

and generates code for the term if no error is reported. If the term is a constant, a `Const` item is returned (and code may not need to be generated); if the term is not a constant, the location of the result is returned as determined by the code generator.

•def `term()`:

`x = factor()`

`while SC.sym in {TIMES, DIV, MOD, AND}:`

`op = SC.sym; getSym();`

`...`

`y = factor() # x op y`

`if x.tp == Int == y.tp and op in {TIMES, DIV, MOD}:`

`if type(x) == Const == type(y): # constant folding`

`if op == TIMES: x.val = x.val * y.val`

`elif op == DIV: x.val = x.val // y.val`

`elif op == MOD: x.val = x.val % y.val`

`else: x = CG.genBinaryOp(op, x, y)`

`elif ...`

`else: mark('bad type')`

`return x`

## ... Code Generation for Expressions ...

---

Procedure `genBinaryOp(op, x, y)` generates code for `x op y` if `op` is PLUS, MINUS, TIMES, DIV, MOD:

```
•def genBinaryOp(op, x, y):
 if op == PLUS: y = put('add', x, y)
 elif op == MINUS: y = put('sub', x, y)
 elif op == TIMES: y = put('mul', x, y)
 elif op == DIV: y = put('div', x, y)
 elif op == MOD: y = put('mod', x, y)
 elif ...
 return y
```

## ... Code Generation for Expressions

---

Procedure `simpleExpression()` generates code for:

`simpleExpression ::= "["+" | "-"] term {"+" | "- | "or"} term}`

```
•def simpleExpression():
 if SC.sym == PLUS: getSym(); x = term()
 elif SC.sym == MINUS:
 getSym(); x = term()
 if x.tp != Int: mark('bad type')
 elif type(x) == Const: x.val = - x.val # constant folding
 else: x = CG.genUnaryOp(MINUS, x)
 else: x = term()
 while SC.sym in {PLUS, MINUS, OR}:
 op = SC.sym; getSym()
 y = term() # x op y
 if x.tp == Int == y.tp and op in {PLUS, MINUS}:
 if type(x) == Const == type(y): # constant folding
 if op == PLUS: x.val = x.val + y.val
 elif op == MINUS: x.val = x.val - y.val
 else: x = CG.genBinaryOp(op, x, y)
 elif ...
 return x
```

## Code Generation for Assignments

---

- `def statement():`
  - ... error handling
  - if `SC.sym == IDENT`:
    - `x = find(SC.val); getSym()`
    - if `type(x) in {Var, Ref}`:
      - `x = CG.genVar(x); x = selector(x)`
      - if `SC.sym == BECOMES`:
        - `getSym(); y = expression()`
        - if `x.tp == y.tp in {Bool, Int}`: `x = CG.genAssign(x, y)`
        - else: `mark('incompatible assignment')`
      - elif `SC.sym == EQ`:
        - `mark(':= expected'); getSym(); y = expression()`
        - else: `mark(':= expected')`
  - `def genAssign(x, y):`
    - if `type(x) == Var`:
      - if `type(y) == Cond...`
      - elif `type(y) != Reg`: `y = loadItem(y); r = y.reg`
      - else: `r = y.reg`
      - `putMemOp('sw', r, x.reg, x.adr); releaseReg(r)`
    - elif ...

## Register Allocation

---

- Rather than using the registers strictly as a stack, we employ a more liberal scheme, which will be of benefit later:
- `GPregs = {'$t0', '$t1', '$t2', '$t3', '$t4', '$t5', '$t6', '$t7', '$t8'}`
- ```
def genProgStart():  
    asm = []; regs = set(GPregs)
```
- ```
def obtainReg():
 if len(regs) == 0: mark('out of registers'); return R0
 else: return regs.pop()
```
- ```
def releaseReg(r):  
    if r in GPregs: regs.add(r)
```

Loading Items into Registers

- If it turns out that delaying the code generation was unsuccessful, the procedure `loadItem(x)` will move the item `x` to a register:
- ```
def loadItemReg(x, r):
 if type(x) == Var:
 putMemOp('lw', r, x.reg, x.adr); releaseReg(x.reg)
 elif type(x) == Const:
 testRange(x); putOp('addi', r, R0, x.val)
 elif type(x) == Reg: # move to register r
 putOp('add', r, x.reg, R0)
```
- ```
def loadItem(x):  
    if type(x) == Const and x.val == 0: r = R0 # use R0 for "0"  
    else: r = obtainReg(); loadItemReg(x, r)  
    return Reg(x.tp, r)
```
- Note that if constant zero is to be loaded into a register, no code is generated at all but rather register 0 is used instead!

Examples for Delayed Code Generation

- $u := 3$
ADDI 1,0,3
STW 1,0,adr(u)
- $u := x + y \times 7$
LDW 1,0,adr(y)
MULI 1,1,7
LDW 2,0,adr(x)
ADD 2,2,1
STW 2,0,adr(u)
- $u := 0$
STW 0, 0, adr(u)
- Note that zero is the most frequent constant, $\approx 20\%$!

Conditionals, Iterations, and Boolean Expressions

- In principle, relational operators ($=$, \neq , $<$, \leq , $>$, \geq) can be treated as arithmetic operations. For example, an instruction EQL x, y could compare variables x and y for equality and put the result true or false (coded by say 0 and 1) on top of the stack. Instruction BT L and BF L would branch to location L depending on whether the top of the stack is true or false, respectively:

if x = y then	EQL x,y
S	BF L
	Code(S)
	L: ...

Similar instructions could be provided for the other relational operators. Unfortunately, typical RISC compare only a register with zero. Our RISC architecture features:

BEQ a,c	if R.a = 0 then PC := PC + c else PC := PC + 1
BNE a,c	if R.a \neq 0 then PC := PC + c else PC := PC + 1
BLT a,c	if R.a < 0 then PC := PC + c else PC := PC + 1
...	

Comparisons with RISC

- Comparisons are done by first subtracting one side from the other, ignoring possible overflow or underflow. The CMP instruction is like SUB, except that it does not trap on overflow or underflow. For example we get (assuming x, y are in registers x, y, respectively):

if x = y then	CMP i,x,y
S	BNE i,L
	Code(S)
	L: ...

Note that the BNE instruction is the negation of that in the condition. This applies to all comparisons.

- The type Item is extended by a new mode Cond and a new field c, which contains the condition according to following numbering:

=	0	≠	1
<	2	≥	3
≤	4	>	5

Relational Expressions ...

- Relational expressions occur in:

expression = SimpleExpression ["=" | "≠" | "<" | "≤" | ">" | "≥"]
SimpleExpression]

- def expression():
 x = simpleExpression()
 while SC.sym in {EQ, NE, LT, LE, GT, GE}:
 op = SC.sym; getSym(); y = simpleExpression() # x op y
 if x.tp == y.tp in (Int, Bool):
 if type(x) == Const == type(y): # constant folding
 if op == EQ: x.val = x.val == y.val
 elif op == NE: x.val = x.val != y.val
 elif op == LT: x.val = x.val < y.val
 elif op == LE: x.val = x.val <= y.val
 elif op == GT: x.val = x.val > y.val
 elif op == GE: x.val = x.val >= y.val
 x.tp = Bool
 else: x = CG.genRelation(op, x, y)
 else: mark('bad type')
 return x

... Relational Expressions

- Procedure `genRelation(op, x, y)` generates code for `x op y` if `op` is `EQ`, `NE`, `LT`, `LE`, `GT`, `GE`. Items `x` and `y` cannot be both constants. A new `Cond` item is returned:
- ```
def genRelation(op, x, y):
 if type(x) != Reg: x = loadItem(x)
 if type(y) != Reg: y = loadItem(y)
 return Cond(op, x.reg, y.reg)
```

- BR L branches unconditionally to L. It is coded as BEQ 0,L.

## Scheme for While Loops

---

- while exp do  
    S  
    L<sub>0</sub>: Code(exp)  
        Bcond L<sub>1</sub>  
        Code(S)  
        BR L<sub>0</sub>  
    L<sub>1</sub>:
- Alternatively, the code for the expression can be placed at the end. This saves the execution of one BR instruction in the more frequent case that the loop is repeated:

```
BR L1
L0: Code(S)
L1: Code(exp)
Bcond L0
```

## Scheme for For Loops

---

- In C and C++, the for statement

for (exp<sub>1</sub>; exp<sub>2</sub>; exp<sub>3</sub>) S

is equivalent to

exp<sub>1</sub>; while (exp<sub>2</sub>) {S; exp<sub>3</sub>;}

Compilers usually make this transformation before code generation.

- In Pascal, the for statement

for v := exp<sub>1</sub> to exp<sub>2</sub> do S

is equivalent to

t<sub>1</sub> := exp<sub>1</sub>; t<sub>2</sub> := exp<sub>2</sub>;

if t<sub>1</sub> <= t<sub>2</sub> then

begin v := t<sub>1</sub>; S;

while v <> t<sub>2</sub> do begin v := v + 1; S end

end

The value of v is undefined at the end of the loop.

Why?

## Scheme for Case Statements

- In the simplest form, a single jump table is used. If that would get too big, compilers either report an error or use a combination of if-then-else and jump tables:

```

case exp of
 c0: S0 ;
 c1: S1 ;
 ...
otherwise
 Sn
end

```

```

Code(exp)
SUBI Ri, cmin
CMPI Ri, cmax
BGT Ln
Code(T[Ri])
Code(PC:=Ri)
L0: Code(S0)
BR L
L1: Code(S1)
BR L
...
Ln: Code(Sn)
L:

```

T:

|                  |                  |
|------------------|------------------|
| c <sub>min</sub> | L <sub>min</sub> |
| ...              | ...              |
| c <sub>i</sub>   | L <sub>i</sub>   |
|                  | L <sub>n</sub>   |
| c <sub>max</sub> | L <sub>max</sub> |



## Code Generation for While Loops

---

- `elif SC.sym == WHILE:`  
    `getSym(); t = CG.genWhile(); x = expression()`  
    `x = CG.genDo(x)`  
    `if SC.sym == DO: getSym()`  
    `else: mark("'do' expected")`  
    `y = statement()`  
    `x = CG.genWhileDo(t, x, y)`
- `def genWhile():`  
    `lab = newLabel()`  
    `putLab(lab)`  
    `return lab`
- `def genDo(x):`  
    `return genThen(x)`
- `def genWhileDo(lab, x, y):`  
    `putInstr('b', lab)`  
    `putLab(x.labA)`

## Code Generation for Conditionals

---

- `elif SC.sym == IF:`  
    `getSym(); x = expression();`  
    `x = CG.genThen(x)`  
    `if SC.sym == THEN: getSym()`  
    `y = statement()`  
    `if SC.sym == ELSE:`  
        `y = CG.genElse(x, y)`  
        `getSym(); z = statement()`  
        `x = CG.genIfElse(x, y, z)`  
    `else:`  
        `x = CG.genIfThen(x, y)`
- `def genThen(x):`  
    `if type(x) != Cond: x = loadBool(x)`  
    `putBranchOp(condOp(negate(`  
        `x.cond)), x.left, x.right, x.labA[0])`  
    `releaseReg(x.left)`  
    `releaseReg(x.right)`  
    `putLab(x.labB)`  
    `return x`
- `def genIfThen(x, y):`  
    `putLab(x.labA)`
- `def genElse(x, y):`  
    `lab = newLabel()`  
    `putInstr('b', lab)`  
    `putLab(x.labA);`  
    `return lab`
- `def genIfElse(x, y, z):`  
    `putLab(y)`

## Boolean Operations

---

- In principle, boolean operators (and, or, not) could be treated as arithmetic operations. However, many programming languages define evaluation of 'and' and 'or' conditionally:

$p \text{ or } q = \text{if } p \text{ then true else } q$

$p \text{ and } q = \text{if } p \text{ then } q \text{ else false}$

As soon as the result is fixed, the remaining parts must not be evaluated. This way, expressions like

$(x \neq \text{nil}) \text{ and } (x.^n = 5)$

$(y = 0) \text{ or } (7 \text{ div } y > 2)$

are allowed. (^ is dereferencing in Pascal; the original Pascal standard did not require conditional evaluation to give compilers more freedom.) Therefore, the same techniques as for conditional statements can be applied;

$\text{if } (x \leq y) \text{ and } (y < z) \text{ then } S$

is compiled the same way as:

$\text{if } x \leq y \text{ then if } y < z \text{ then } S$

## Boolean AND

---

- Code pattern for AND:

if (a < b ) and (c ≤ d) and (e > f)  
then S<sub>0</sub>  
else S<sub>1</sub>

```
CMP a,b
BGE L1
CMP c,d
BGT L1
CMP e,f
BLE L1
Code(S0)
BR L
L1: Code(S1)
L:
```

All conditions  
are negated.

## Boolean OR

---

- Code pattern for OR:

if (a < b ) or (c ≤ d) or (e > f)  
then S<sub>0</sub>  
else S<sub>1</sub>

```
CMP a,b
BLT L0
CMP c,d
BLE L0
CMP e,f
BLE L1
L0: Code(S0)
BR L
L1: Code(S1)
L:
```

———— Last condition  
is negated.

## Boolean NOT ...

---

- No code needs to be generated for NOT, only branch locations get exchanged or branch conditions negated. For example:

if not ((a < b ) and (c ≤ d) and (e > f))  
then S<sub>0</sub>  
else S<sub>1</sub>

```
CMP a,b
BGE L0
CMP c,d
BGT L0
CMP e,f
BGT L1
L0: Code(S0)
BR L
L1: Code(S1)
L:
```

Branch now to L<sub>0</sub> instead of L<sub>1</sub>.

BGT instead of BLE.

## ... Boolean NOT

---

if not ((a < b ) or (c ≤ d) or (e > f))  
then S<sub>0</sub>  
else S<sub>1</sub>

CMP a,b  
BLT L<sub>1</sub>      Branch now to  
CMP c,d      L<sub>1</sub> instead of L<sub>0</sub>.  
BLE L<sub>1</sub>  
CMP e,f  
BGT L<sub>1</sub>      BGT instead  
                 of BLE.  
L<sub>0</sub>: Code(S<sub>0</sub>)  
BR L  
L<sub>1</sub>: Code(S<sub>1</sub>)  
L:

## Code Generation for Boolean Operators

---

- For arithmetic operators, the code for the operator is generated after the code for both operands.
- For “and” and “or”, a branch instruction is instead generated after each operand:
  - for “and”, this results in a list of forward branches to the location with the code if the condition is false;
  - for “or”, this results in a list of forward branches to the location with the code if the condition is true.

These forward branches are kept as a list in Cond items. They are emitted as soon the location is known:

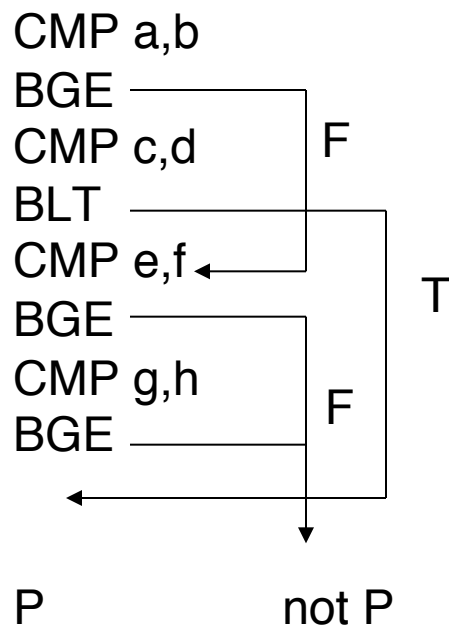
- either at the end of the boolean expression, or
- when “and” and “or” are mixed.



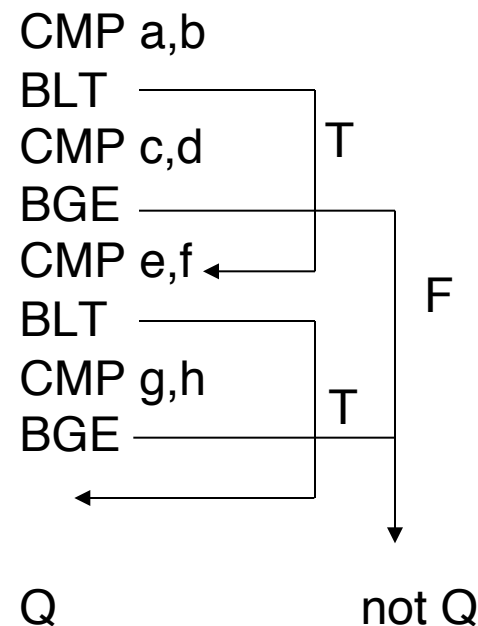
## Mixing AND and OR

- Examples:

$P = ((a < b) \text{ and } (c < d)) \text{ or } ((e < f) \text{ and } (g < h))$



$Q = ((a < b) \text{ or } (c < d)) \text{ and } ((e < f) \text{ or } (g < h))$



## Boolean Assignment

---

- For a boolean expression  $p$ , the assignment  
     $x := p$   
is compiled the same way as  
    if  $p$  then  $x := \text{true}$  else  $x := \text{false}$
- For example, the pattern for  $q := x < y$  is  
    CMP x,y  
    BGE L<sub>1</sub>  
    ADDI 1,0,1  
    BR L  
L<sub>1</sub>: ADDI 1,0,0  
L: STW 1,q
- However, the assignment of boolean variables is not treated as a condition. Hence  $q := p$  results in the expected code sequence:  
    LDW 1,0,p  
    STW 1,0,q

## Short-Circuit Evaluation

---

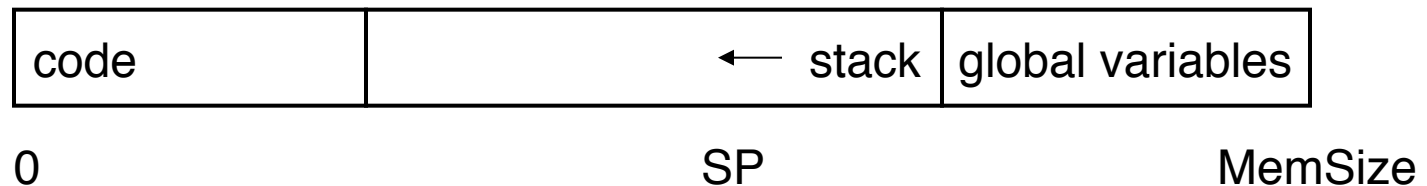
- In the scheme presented, no instructions for “and”, “or”, “not” are generated, unlike for arithmetic expressions, where instructions corresponding to +, −, \*, div, mod are generated. During evaluation of a boolean expression, its value is not given by the contents of a register but by the position in the code.
- This scheme allows evaluation of boolean expressions to be stopped as soon as the result is determined. This is called short-circuit evaluation.

## Procedures and Locality

---

- Procedures may be called from different places in the code, and may themselves call other procedures. The return addresses have to be stored according to the first-in first-out principle on a stack.
- For languages which allow recursion, the maximal size of the stack is not known at compile time. Therefore, main memory is used for the stack.

Typical memory layout:

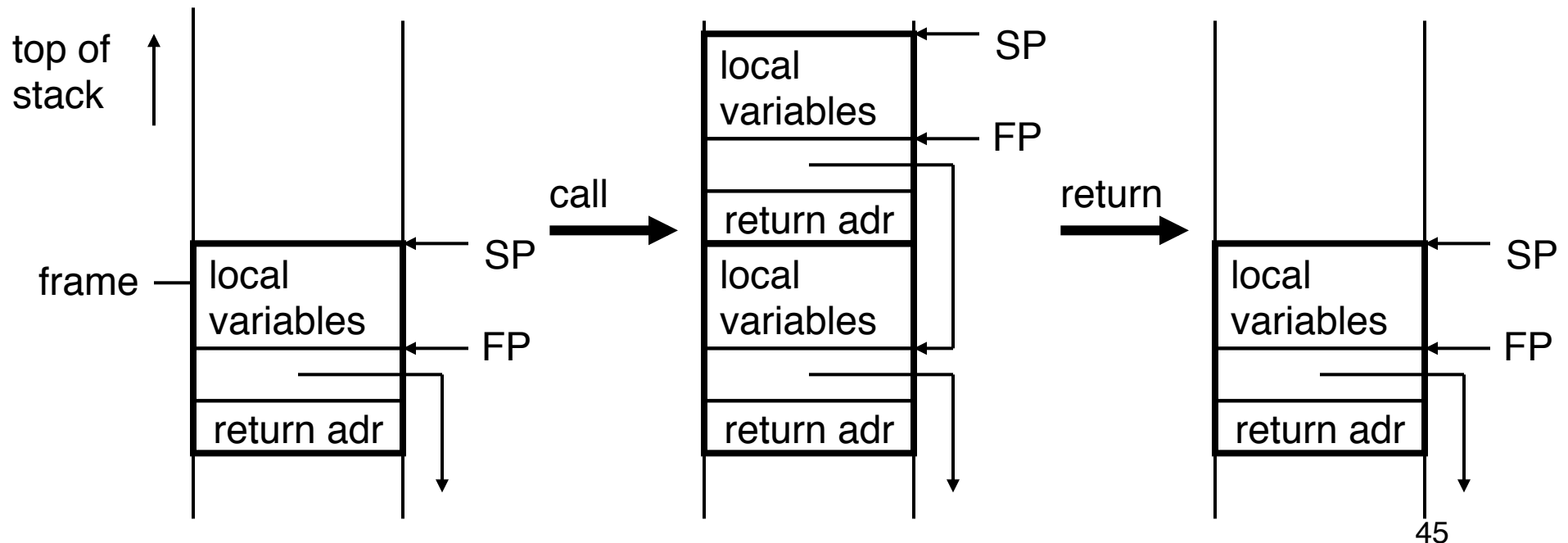


- Calling a procedure involves the stack pointer SP (R30 by convention) and link register LNK (R31 in RISC).

|                    |                  |                         |
|--------------------|------------------|-------------------------|
| Procedure Call     | L: BSR c         | R31:=(PC+1)×4; PC:=PC+c |
| Procedure Prologue | L+c:PSH LNK,SP,4 | R30:=R30-4; M[R30]:=R31 |
| Procedure Epilogue | POP LNK,SP,4     | R31:=M[R30]; R30:=R30+4 |
|                    | RET 0,0,LNK      | PC:=R31                 |

## Activation Frame

- Local variables of procedures are also placed on the stack, meaning that two procedures which are not called simultaneously may use the same memory. (An exception are static variables in C and Fortran.)
- The storage block with local variables, return address, and other data is called the procedure activation record or activation frame. The frame pointer is a register, here R29 by convention, which points to the current frame. Local variables are accessed relative to the frame pointer.



## Procedure Prologue and Epilogue with Local Variables

---

- Suppose the local variables occupy  $n$  bytes.

|                    |              |                      |
|--------------------|--------------|----------------------|
| Procedure Prologue | PSH LNK,SP,4 | SP:=SP-4; M[SP]:=LNK |
|                    | PSH FP,SP,4  | SP:=SP-4; M[SP]:=FP  |
|                    | ADD FP,0,SP  | FP:=SP               |
|                    | SUBI SP,SP,n | SP:=SP-n             |
| Procedure Epilogue | ADD SP,0,FP  | SP:=FP               |
|                    | POP FP,SP,4  | FP:=M[SP]; SP:=SP+4  |
|                    | POP LNK,SP,4 | LNK:=M[SP]; SP:=SP+4 |
|                    | RET 0,0,LNK  | PC:=R31              |

- In case the size of local variables is known at compile time (as in Pascal0 so far), only one pointer, SP, is necessary. Local variables can then be addressed relative to SP.
- In case the size of local variables is unknown (or the size of the parameters passed is unknown), both SP and FP are necessary.
- Processors with "complex" instructions like Motorola 68000 combine the prologue and epilogue into a single instruction.

Why not  
RISC?

## Addressing of Variables ...

---

- Global variables: They are typically addressed by absolute addressing. Since the size of the available memory is not known at compile time, (negative) addresses relative to zero are generated during compilation and incremented by the memory size after loading. This is called relocation. In the Pascal0 compiler, relocation information is displayed after compilation.
- Local variables of current procedure: These are addressed relative to FP. Instructions LDW a,b,c and STW a,b,c provide the possibility to specify an offset, e.g.

LDW R1,FP,offset(x)

$R1 := M[FP + \text{offset}(x)]$

How efficient is  
access to globals  
and locals?

## ... Addressing of Variables

---

- Local variables at intermediate level: Consider as an example:

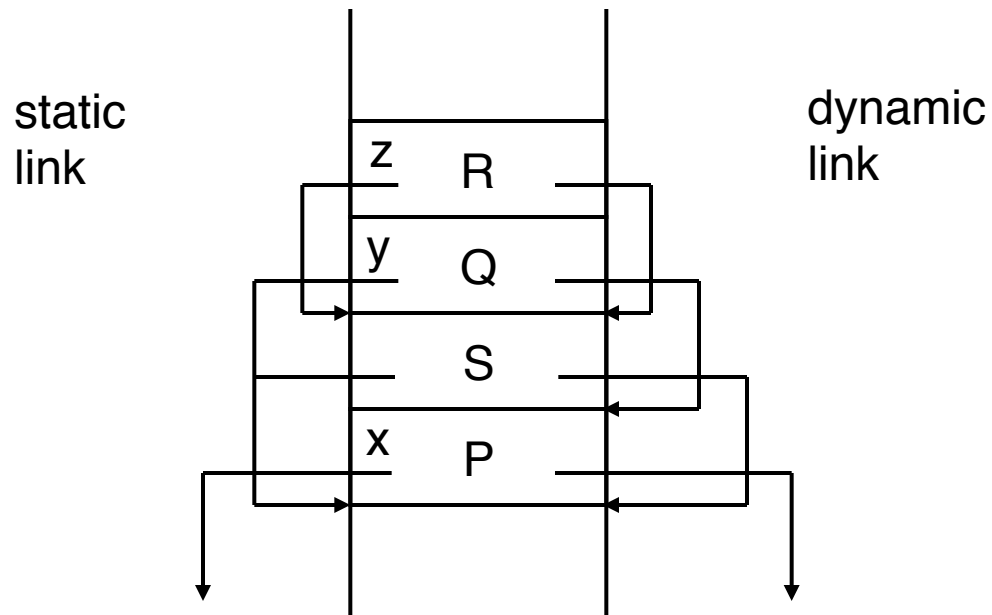
```
var w: integer
procedure P
 var x: integer
 procedure Q
 var y: integer
 procedure R
 var z: integer
 x := y + z
 R
 procedure S
 Q
 Q;S
program M
P
```



## Static Link vs. Dynamic Link ...

---

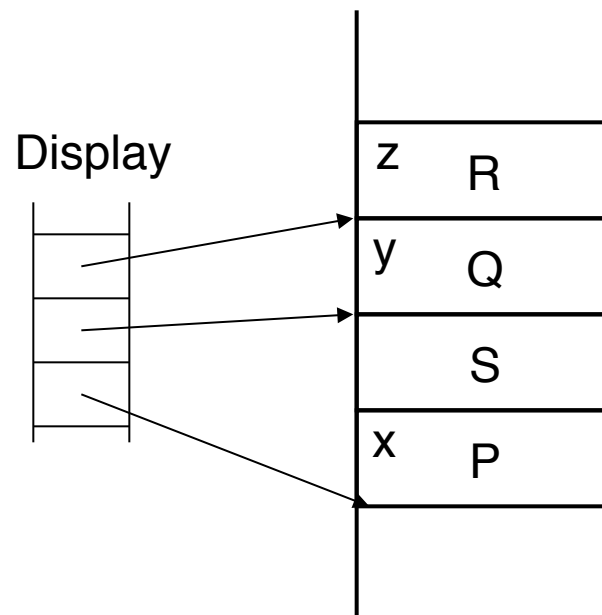
- For local variables at intermediate levels, their address depends on the chain of procedure calls, e.g.  
 $P \rightarrow Q \rightarrow R$  and  $P \rightarrow S \rightarrow Q \rightarrow R$
- The address of intermediate variables cannot be obtained by following the FP links with a fixed number of steps. In addition to the dynamic link which roots in FP, a further static link has to be kept.



## ... Static Link vs. Dynamic Link

---

- This implies that access to intermediate variables requires several steps and is rather inefficient.
- Other solutions involve keeping a stack with pointers to the local variables at each level of nesting. Since the maximal size of the stack is the level of nesting, it is known at compile time. This stack, implemented as an array, is called the display. Access with a display takes a constant number of instructions, but requires additional bookkeeping.



Since access to intermediate variables is rather infrequent, the choice of the solution is not critical. In fact, the extra effort for the display may make it less efficient.

## Parameter Passing Mechanisms ...

---

- Procedures are declared with formal parameters. On a procedure call, these are substituted by actual parameters. Different kinds of parameter substitution:
- Value parameter (in parameter): the actual parameter can be an arbitrary expression. The expression is evaluated before the call and its value is passed to the procedure.
- Result parameter (out parameter): the actual parameter must be a variable, also with indexing and field selection. The formal parameter is undefined on procedure entry and must assigned a value before procedure exit. The value is then assigned to the actual parameter.
- Value-Result parameter (in-out parameter): both a value and result parameter.
- Reference parameter (variable parameter): the actual parameter must be a variable, also with indexing and field selection. The address of the variable is passed. The formal parameter refers constantly to the actual parameter through its address.

## ... Parameter Passing Mechanisms

---

- Name parameter: the actual parameter is evaluated each time the formal parameter refers to it. This way, a procedure call can be explained by textual substitution. Only used in Algol 60, since it leads to inefficiency.

What is the output of following code, with different parameter passing mechanism?

```
procedure P(x: integer)
 i:=2; write(x); x:=7
```

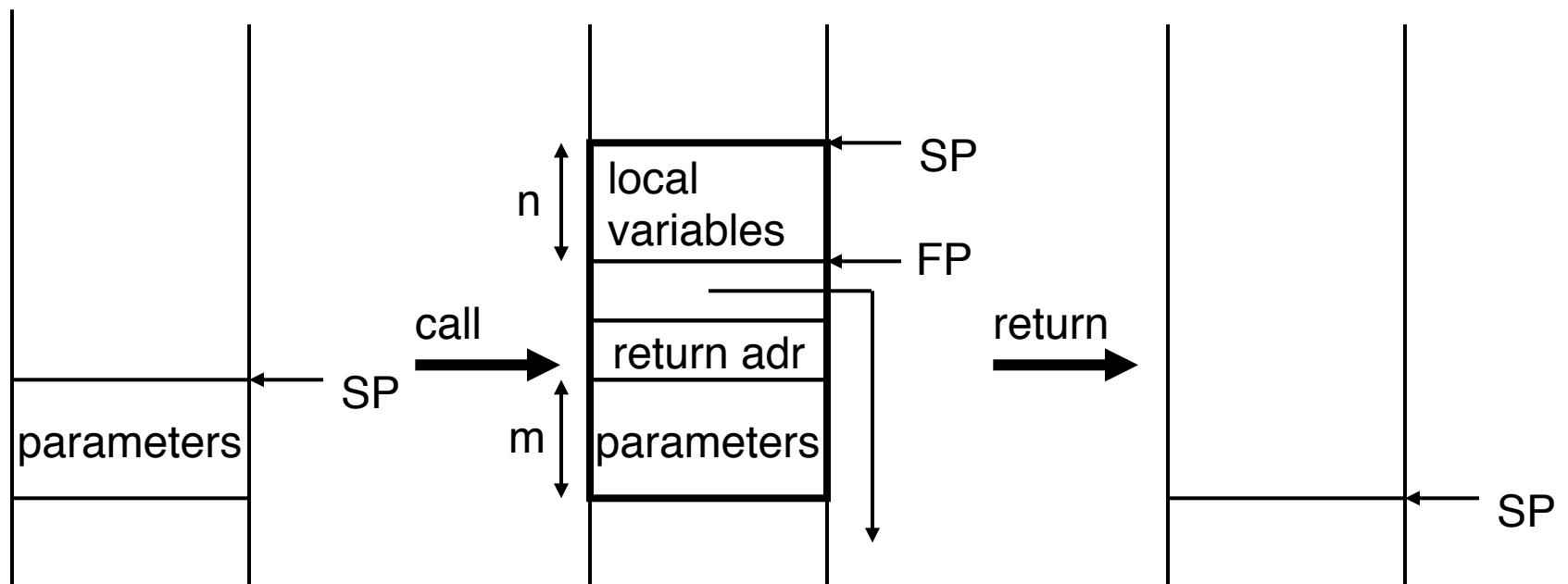
```
a[1]:=1; a[2]:=5; i:=1; P(a[i]); write(a[1],a[2])
```

- Note that some parameter passing mechanisms may lead to aliasing between parameters or between parameters and global variables.

Which and how?

## Code for Parameter Passing ...

- Pascal0, as Pascal and C, has value and reference parameters. Value parameters are evaluated and addresses of the reference parameters are determined and pushed on the stack before the procedure call (BSR).
- In the procedure body, parameters are accessed like local variables relative to FP, but now with a positive offset.



### ... Code for Parameter Passing ...

---

- Value and reference parameters are placed on the stack with PSH:

PSH x, SP,4                      SP:=SP-4; M[SP]:=x

- The epilogue removes the parameters at exit from the procedure:

ADD SP,0,FP                      SP:=FP

POP FP,SP,4                      FP:=M[SP]; SP:=SP+4

POP LNK,SP,m+4                  LNK:=M[SP]; SP:=SP+m+4

RET 0,0,LNK                      PC:=LNK

- The FP relative address of parameter  $p_i$  is given by:

$$\text{adr}(p_i) = \text{size}(p_{i+1}) + \dots + \text{size}(p_n) + 8$$

This implies that the address of parameters can be determined only after all parameters have been parsed. Value parameters can vary in size, reference parameters take one word.

### ... Code for Parameter Passing

---

- Reference parameters require one indirection. Since there is no indirect addressing mode in RISC, the address is first loaded in a register with LDW.
- In the compiler, this is treated by emitting the LDW instruction and then setting the mode of the corresponding item to VarClass, like for any variable.
- Parsing a procedure leads to opening a new scope with the formal parameters and local variables.

```
 procedure P(x,y: integer)
 var z:integer
```

```
 ...
```

At the end of P, the scope is closed but a pointer to the list of parameters is still retained in the dsc field of the symbol table entry for P.

## Standard Procedures

---

- Standard procedures are predeclared and hence can be called from anywhere. Examples:
  - arithmetic functions: abs, sin, ...
  - type conversions: round, ord, ...
  - input-output procedures: read, write, open, ...
- For many of these procedures, code is generated in-line, rather than making a procedure call. For example in Pascal0:  

|          |                |
|----------|----------------|
| write(x) | LDW 1,0,adr(x) |
|          | WRD 0,0,1      |
- Some programming languages allow the user to declare a procedure as in-line for efficiency. Some standard procedure are treated as normal procedure calls to standard libraries, e.g. write in Pascal. Some standard procedures require special instructions, e.g. arithmetic functions for a special arithmetic processor.



## Function Procedures ...

---

- Example:  
    procedure sign (x: integer) → (s: integer)  
        if x > 0 then s := 1 else ...
- Return value can be passed back in a register: efficient, convenient if the result need to be in a register anyway, e.g. in

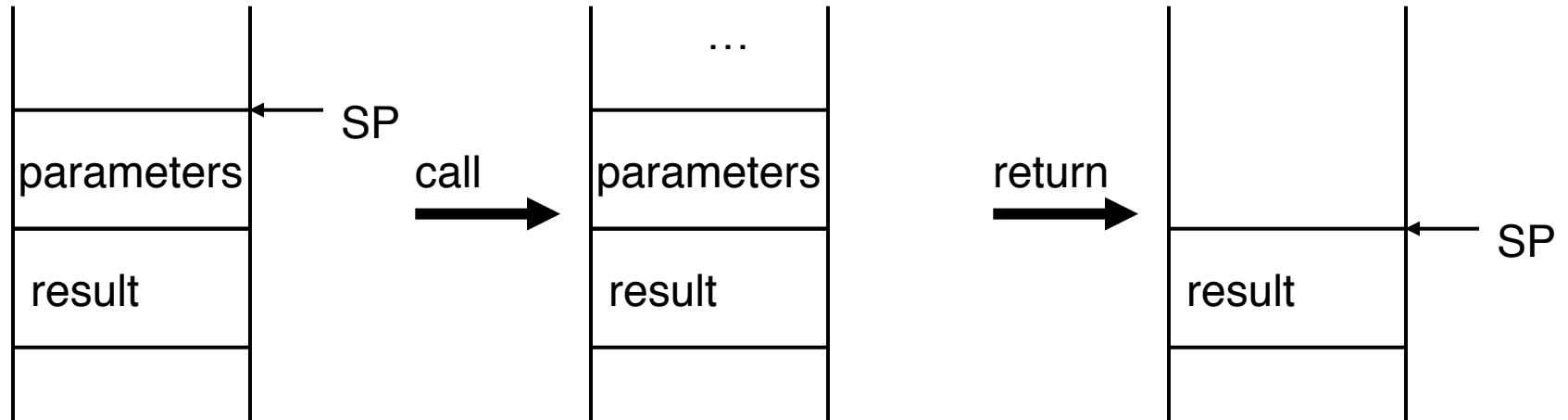
    sign(y) × z

Only works if the result can fit into a single or a few registers. Pascal allows basic types but no records or arrays.

## ... Function Procedures

---

- Return value can be passed back on a stack. Space is reserved before the actual parameters are pushed on the stack:



- Address of the result is first pushed on the stack. Within the procedure, the result is treated like a reference parameter. This eliminates the need for large copy operations, e.g. in  
     $\text{matrixA} \leftarrow \text{transpose}(\text{matrixB})$   
but makes selection or indexing of the result difficult or impossible, e.g. in:  
     $x \leftarrow \text{transpose}(\text{matrixB})[i,j]$

## Open Arrays ...

---

- A formal array parameter whose length is unknown at compile time is called an open array (conformant array in Pascal).
- For checking index bounds and in order to provide a way for determining the actual length, the length is passed as well. Example:

```
procedure writeString(s: char[])
 for i := 0 to length(s) - 1 do writeChar(s[i])
```

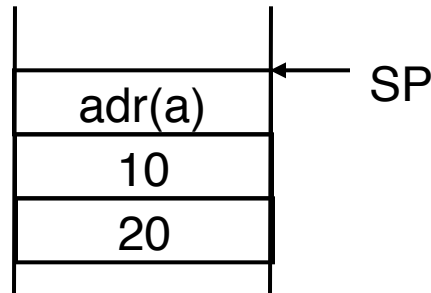
- In the case of multidimensional arrays, the length is also needed for determining the element addresses.

```
procduce print(var A: integer [] [])
 write(A[i,j])
```

## ... Open Arrays

---

- In this example, an array descriptor with the length in each dimension and the address is pushed on the stack before the call, e.g. for P(a) if a: array 10 of array 20 of integer:



- In the case of a value parameter, the copying of the parameter on the stack is better done only once within the procedure after the prologue rather than at each procedure call. Space for the copy is allocated on the top of the stack by decrementing SP. In this situation, the distinction of SP and FP becomes necessary.
- The scheme of passing descriptors rather than values can be used for any types with size unknown at compile time, like extensible records (objects).