Compiling Simple Assignments

Moo y Sagiv
Te Aviv University
`sagiv@math.tau.ac.il`
and
Reinhard Wi he m
Universität des Saar andes
`wilhelm@cs.uni-sb.de`

Apri 21, 1997

Subjects

- Compi e vs Run Time nformation
- L va ues vs R va ues
- The P machine
- Code generation for expressions and assignments

Compile- vs. Run-Time Information

compile-time-information (static): nformation contained in or derivab e from the source program

- The source code
- The types of variab es (in most anguages)
- The scope of variab es
- The va ues of constants
- The addresses of static variab es
- The re ative addresses of automatic variab es
- The size of static data (sca ars, stat arrays, records)
-

run-time-information (dynamic): nformation on y avai ab e at run time

- The va ues of variab es
- The va ues of conditions
- The depth of recursion
- The size of dynamic data (dyn arrays, ists, trees)
-

L-values vs. R-values

- Assignment $x := exp$ is compi ed into:
  1. Compute the **address** of x
  2. Compute the **value** of exp
  3. Store the va ue of exp at the address of x
- Genera ization

  R-value

  r va (x) $=$ va ue of x

  r va (5) $=$ 5

  r va $(x+y)$ $=$ r va (x) $+$ r va (y)

  L-value

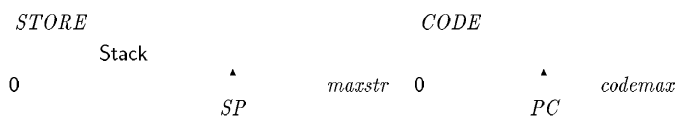
  va (x) $=$ address of x

  va (5) $=$ **undefined**

  va $(x+y)$ $=$ **undefined**

  va $(a[i])$ $=$ va (a) $+ \text{ some function of }$ r va (i)

The P-Machine



- Memory
- nstructions operate on the stack
- Typed instructions (using Pasca ordina types)

$+_i$	1	adds the two top most int va ues on the stack;
	2	removes these from the stack;
	3	stores (pushes) the resu t on the stack

- Operand types in the P machine

i	integer
r	rea
a	address
b	boo ean

- Type indications in the instruction definitions

T	a types
N	numerica types

P-machine main loop

```
**while true do begin**
        $PC := PC + 1$ ;
        execute instruction in  ocation $CODE[PC - 1]$
**end**;
```

Why PC increment before instruction execution?
Reason: jumps and procedure ca s

Example Program

Pascal-program

```
program foo(input, output) ;
var x, y : integer;
begin
        x := 3 ;
        y := x + 7
end.
```

P-code-program

```
  p          initia ization code
  ssp   8                             a  ocate stack space
  sep   3                             space for intermediate computations
  ldc   a   5                         pushes the address of x
  ldc   i   3                         pushes 3
  sto   i                             stores 3 in x
  ldc   a   6                         pushes the address of y
  ldo   i   5                         pushes the va ue of x
  ldc   i   7                         pushes 7
  add   i                             pushes x + 7
  sto   i                             stores x+7 in y
  stp
```

The definition of P-instructions

Instruction	Meaning	Condition	Result

Instruction: name of the instruction and ist of its parameters,

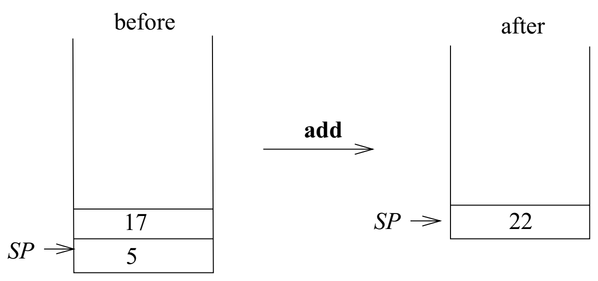
Meaning: program in a very reduced imperative anguage consisting of assignments between machine ressources and conditiona s,

Condition: condition on the execution of the instruction, often a pattern describing the expected contents of the top end of the stack, i e , the types of the ce contents,

Result: description of the resu t, a pattern describing the resu ting stack contents

P-instructions for Arithmetic

Instr.	Meaning	Cond.	Res.
add N	$STORE[SP \quad 1] := STORE[SP \quad 1] +_N STORE[SP]$;	$\begin{pmatrix} N \\ N \end{pmatrix}$	(N)
	$SP := SP \quad 1$		
sub N	$STORE[SP \quad 1] := STORE[SP \quad 1] \quad _N STORE[SP]$;	$\begin{pmatrix} N \\ N \end{pmatrix}$	(N)
	$SP := SP \quad 1$		
mul N	$STORE[SP \quad 1] := STORE[SP \quad 1] *_N STORE[SP]$;	$\begin{pmatrix} N \\ N \end{pmatrix}$	(N)
	$SP := SP \quad 1$		
div N	$STORE[SP \quad 1] := STORE[SP \quad 1] /_N STORE[SP]$;	$\begin{pmatrix} N \\ N \end{pmatrix}$	(N)
	$SP := SP \quad 1$		
neg N	$STORE[SP] := \quad _N STORE[SP]$	(N)	(N)



P-instructions for Boolean Operations

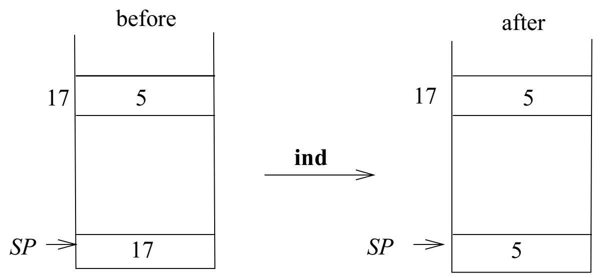
Instr.	Meaning	Cond.	Res.
and	$STORE[SP \quad 1] := STORE[SP \quad 1]$ *and* $STORE[SP]$;	$\begin{pmatrix} b \\ b \end{pmatrix}$	(b)
	$SP := SP \quad 1$		
or	$STORE[SP \quad 1] := STORE[SP \quad 1]$ *or* $STORE[SP]$;	$\begin{pmatrix} b \\ b \end{pmatrix}$	(b)
	$SP := SP \quad 1$		
not	$STORE[SP] :=$ *not* $STORE[SP]$	(b)	(b)

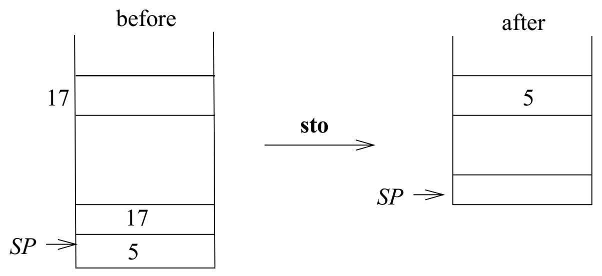
P-instructions for comparisons

Instr.	Meaning	Cond.	Res.
equ T	$STORE[SP \quad 1] := STORE[SP \quad 1] =_T STORE[SP]$; $SP := SP \quad 1$	$\binom{T}{T}$	(b)
geq T	$STORE[SP \quad 1] := STORE[SP \quad 1] \geq_T STORE[SP]$; $SP := SP \quad 1$	$\binom{T}{T}$	(b)
leq T	$STORE[SP \quad 1] := STORE[SP \quad 1] \leq_T STORE[SP]$; $SP := SP \quad 1$	$\binom{T}{T}$	(b)
les T	$STORE[SP \quad 1] := STORE[SP \quad 1] <_T STORE[SP]$; $SP := SP \quad 1$	$\binom{T}{T}$	(b)
grt T	$STORE[SP \quad 1] := STORE[SP \quad 1] >_T STORE[SP]$; $SP := SP \quad 1$	$\binom{T}{T}$	(b)
neq T	$STORE[SP \quad 1] := STORE[SP \quad 1] \neq_T STORE[SP]$; $SP := SP \quad 1$	$\binom{T}{T}$	(b)

P-instructions for load/store

Instr.	Meaning	Cond.	Res.
ldo $T\,q$	$SP := SP + 1$; $STORE[SP] := STORE[q]$	$q \in [0, maxstr]$	(T)
ldc $T\,q$	$SP := SP + 1$; $STORE[SP] := q$	$Typ(q) = T$	(T)
ind T	$STORE[SP] := STORE[STORE[SP]]$	(a)	(T)
sro $T\,q$	$STORE[q] := STORE[SP]$; $SP := SP \quad 1$	(T) $q \in [0, maxstr]$	
sto T	$STORE[STORE[SP \quad 1]] := STORE[SP]$; $SP := SP \quad 2$	$\binom{a}{T}$	





Code Generation

- Assumptions about the input program:
  - No errors (syntax, types)
  - Structure and type information is avai ab e
  - No procedures (for now)
- $\rho(v)$ is the re ative address of program variab e v
- nvariant I_0 about the state of the P machine:
  Let $code_R\ e\ \rho = is$,
   et sp be the va ue of SP before the execution of is
  The execution of is wi eave the va ue of e in $STORE[sp+1]$, and SP's va ue wi be $sp+1$
  $STORE$ is otherwise unchanged

The translation of assignments and expressions

Function		Condition
$code_R(e_1 = e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **equ** T	$Typ(e_1) = Typ(e_2) = T$
$code_R(e_1 \neq e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **neq** T	$Typ(e_1) = Typ(e_2) = T$
$code_R(e_1 + e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **add** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(e_1 \quad e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **sub** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(e_1 * e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **mul** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(e_1 / e_2)\ \rho$	$= code_R\ e_1\ \rho; code_R\ e_2\ \rho;$ **div** N	$Typ(e_1) = Typ(e_2) = N$
$code_R(\quad e)\ \rho$	$= code_R\ e\ \rho;$ **neg** N	$Typ(e) = N$
$code_R\ x\ \rho$	$= code_L\ x\ \rho;$ **ind** T	x is a variab e of type T
$code_R\ c\ \rho$	$=$ **ldc** T c	c is a constant of type T
$code(x := e)\ \rho$	$= code_L\ x\ \rho;\ code_R\ e\ \rho;$ **sto** T	$Typ(e) = T$, x is a variab e
$code_L\ x\ \rho$	$=$ **ldc** a $\rho(x)$	x is a variab e

Correctness of the code generation scheme

Proof by induction of the invariant I_0

Base cases:
$code_R\ x\ \rho$ and $code_R\ c\ \rho$

Inductive step:
Examp e: $code_R(e_1{+}e_2)\,\rho = code_R\,e_1\,\rho; code_R\,e_2\,\rho;$ **add** $=$ is
Let sp be the va ue of SP before the execution of is
nd Assumptions:

- The execution of $code_R\ e_1\ \rho$ eaves the va ue of e_1 in $STORE[SP+1]$ and SP has va ue $sp+1$
- The execution of $code_R\ e_2\ \rho$ eaves the va ue of e_2 in $STORE[SP+2]$ and SP has va ue $sp+2$

Step:

- **add** adds the topmost va ues and eaves the resu t, i e the va ue of e in $STORE[SP+1]$ and SP has va ue $sp+1$

Example

Assume that $Typ(x) = int$ and $\rho(x) = 5$

$code(x := 3)\ \ \rho$
$= code_L\ x\ \rho;\ code_R\ 3\ \rho;$ **sto** i
$=$ **ldc** a 5; $code_R(3)\ \rho$; **sto** i
$=$ **ldc** a 5; **ldc** i 3; **sto** i

Example 2.1

Assume that $\rho(a) = 5$, $\rho(b) = 6$, and $\rho(c) = 7$ and that

$Typ(a) = Typ(b) = Typ(c) = Typ(b{*}c) = Typ(b{+}b{*}c) = int$

$code(a := (b + (b * c)))\ \rho$
$=\ code_L\ a\ \rho$; $code_R\ (b + (b * c))\ \rho$; **sto i**
$=$ **ldc** a 5; $code_R(b + (b * c))\ \rho$; **sto i**
$=$ **ldc** a 5; $code_R(b)\ \rho$; $code_R(b * c)\ \rho$; **add** i; **sto** i
$=$ **ldc** a 5; $code_L(b)\ \rho$; **ind** i; $code_R(b * c)\ \rho$; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; $code_R(b * c)\ \rho$; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; $code_R(b)\ \rho$; $code_R(c)\ \rho$; **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; $code_L(b)\ \rho$; **ind** i; $code_R(c)\ \rho$; **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; $code_R(c)\ \rho$; **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; $code_L(c)\ \rho$; **ind** i; **mul** i; **add** i; **sto** i
$=$ **ldc** a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; **ldc** a 7; **ind** i; **mul** i; **add** i; **sto** i

Interpretation of the generated code

q

```
ssp 9
sep 4
ldc a 5
ldc a 6
ind i
ldc a 6
ind i
ldc a 7
ind i
mul i
add i
sto i
```

(Non-)Optimality of the generated code

- Examp e 1: $a := b$
  - Generated code:
    ldc a $\rho(a)$; **ldc** a $\rho(b)$; **ind** i; **sto**
  - Minima code
    ldo i $\rho(b)$; **sro** i $\rho(a)$
- Examp e 2: $a := a * a;$
  - Generated code:
    ldc a $\rho(a)$
    ldc a $\rho(a)$
    ind i
    ldc a $\rho(a)$
    ind i
    mul i
    sto
  - Minima code
    ldo i $\rho(a)$
    dpl i
    mul i
    sro i $\rho(a)$

Summary

- An inductive definition of the generated code
- Assuming that the re ative addresses of variab es are known at compi e time
- The stack machine simp ifies the task of hand ing comp ex expressions

Compiling Control Flow Statements

Moo y Sagiv
Te Aviv University
sagiv@math.tau.ac.il
and
Reinhard Wi he m
Universität des Saar andes
wilhelm@cs.uni-sb.de

Apri 21, 1997

Subjects

- Syntax of contro flow statements
- Sequence of statements
- Conditiona Statements
- Case statements
- terative Statements

Syntax

Sequence of Statements $st_1; st_2; \cdots; st_n$

Conditional Statements

- **if** e **then** st **fi**
- **if** e **then** st **else** st **fi**
- **case** e **of**

  0: st_1 ;
  1: st_2 ;

  k : st_k
  end

Iterative Statements

- **while** e **do** st **od**
- **repeat** st **until** e

Sequence of statements

- $code(st_1; st_2)\rho = code(st_1)\rho; code(st_2)\rho$
  contro drops through from the ast instruction generated for st_1 to the first instruction generated for st_2
  st_1 and st_2 are compi ed in the same ρ

- Examp e: Assume that $Typ(x) = Typ(y) = int$

  $code(x := 3; y := x + 1)\ \rho$
  $= code(x := 3)\ \rho$; $code(y := x + 1)\ \rho$
  $= code_L\ x\ \rho$; $code_R\ 3\ \rho$; **sto** i ; $code\ (y := x + 1)\ \rho$
  $=$ **ldc** a $\rho(x)$; $code_R(3)\ \rho$; **sto** i $code\ (y := x + 1)\ \rho$
  $=$ **ldc** a $\rho(x)$; **ldc** i 3; **sto** i $code\ (y := x + 1)\ \rho$
  $=$ **ldc** a $\rho(x)$; **ldc** i 3; **sto** i $code_L\ y\ \rho$; $code_R\ (x + 1)\ \rho$; **sto** i
  $=$ **ldc** a $\rho(x)$; **ldc** i 3; **sto** i **ldc** a $\rho(y)$; $code_R\ (x + 1)\ \rho$; **sto** i

Conditional Statements

Machine ressources to imp ement contro statements: unconditiona and conditiona jumps

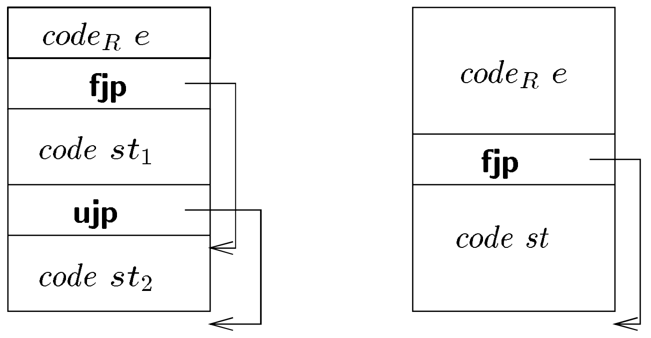
P-code Instructions for Branches

Inst.	Meaning	Cond.	Result
ujp q	$PC := q$	$q \in [0, codemax]$	
fjp q	**if** $STORE[SP] = false$	(b)	$()$
	then $PC := q$	$q \in [0, codemax]$	
	fi;		
	$SP := SP - 1$		

fjp consumes the boo ean va ue on top of the stack

Conditional Statements

- $code(\textbf{if}\ e\ \textbf{then}\ st\ \textbf{fi})\ \rho = code_R\ e\ \rho; \textbf{fjp}\ l; code\ st\ \rho;\ l:$
- $code(\textbf{if}\ e\ \textbf{then}\ st_1\ \textbf{else}\ st_2\ \textbf{fi})\ \rho =$ $code_R\ e\ \rho;\ \textbf{fjp}\ l_1;\ code\ st_1\ \rho;\ \textbf{ujp}\ l_2;\ l_1:$ $code\ st_2\ \rho;\ l_2:$



Conditionals

doublesided onesided

Problem: Forward references to abe s

Solutions:

- Generate symbo ic abe s in assemb y code,
- 2 pass code generation,
- "Back patching" target addresses when known

Examples

Examp e1:
if $a > b$ **then** $c := a$ **else** $c := b$ **fi**

Examp e2:
if $x < 5$ **then** **if** $y < 7$ **then** $c := a$ **fi fi**

Case Statement

Restricted form: nitia section of the natura numbers

Input

```
case $e$ of
        0: $st_1$ ;
        1: $st_2$ ;

        $k$ : $st_k$
end
```

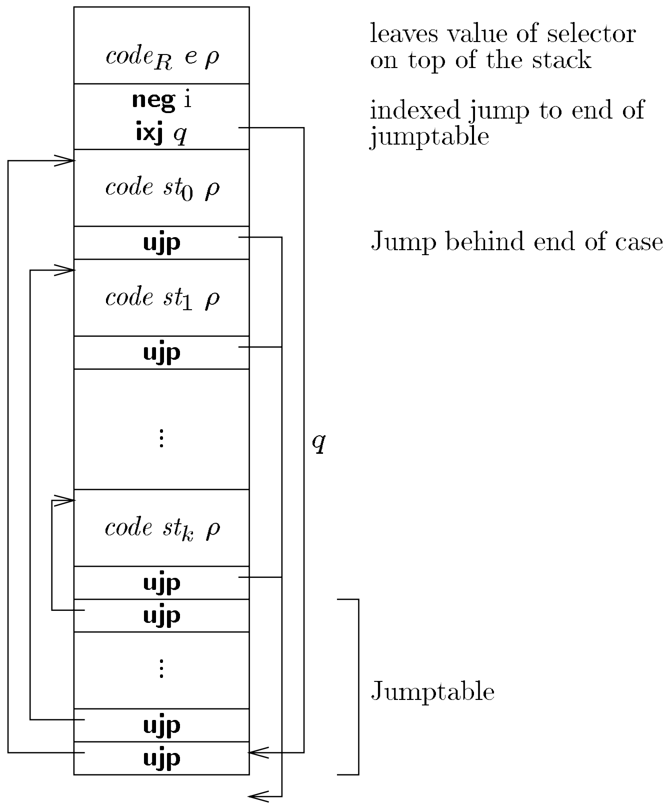
Two Solutions

- Generate nested conditiona s
  if $e = 0$ **then** st_1
  else if $e = 1$ **then** st_2

  else if $e = n$ **then** st_k
  else run time error: unmatched case abe **fi fi** $\cdots$ **fi**
- Jump tab e: $PC :=$ tab e(e)

P-Code Instructions for Jump Table

Instr.	Meaning	Cond.	Result
ixj q	$PC := STORE[SP] + q;$ $SP := SP - 1$	(i)	$()$



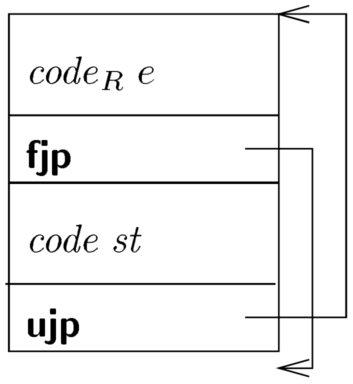
Iterative Statements

- $code$ (**while** e **do** st **od**) $\rho =$
  $l_1 :\ code_R\, e\, \rho;$ **fjp** $l_2;$ $code\ \ st\, \rho;$ **ujp** $l_1;$ $l_2 :$

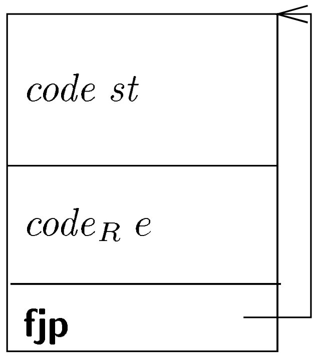
- Example:

  $x := 1;$ **while** $x < 4$ **do**
  $x := x + 1$ **od**

- $code$ (**repeat** st **until** e) $\rho =$
  $l :\ code\ \ st\ \rho;\ code_R\, e\, \rho;$ **fjp** l



while e **do** st **od**



repeat st **until** e

Printed with FinePrint - purchase at www.fineprint.com

Memory Allocation

Mooly Sagiv
Tel Aviv University
sagiv@math.tau.ac.il
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

April 24, 1997

Subjects

- Fixed size data
- Static aays
- Dynamic" Arrays
- Records
- Pointers

Compile-Time Information on Variables

- Name
- Type
- Dimension of arrays
- Selectors in records
- Scope — when is it recognized
- Size — How many bytes are required at run-time to represent the object

Assumptions

- Code generation for a single procedure
- The first 5 stack locations are reserved; they will later be needed for procedure organization.

The function ρ

- For every data type t
  $size(t)$ is the number of bytes required at run-time to represent objects of (static) type t
- Assumption:
  $size(integer) = size(real) = size(char) = size(boolean) = 1$
- Memory allocation for variables in order of appearance
  var $v_0 : t_0; v_1 : t_2; \ldots ; v_k : t_k;$

$$\rho(v_i) = 5 + \sum_{j=0}^{i-1} size(t_j)$$

- Example: **var** $x, y : \mathbf{real}; z : \mathbf{boolean}$

$$\rho = [x \mapsto 5, y \mapsto 6, z \mapsto 7]$$

Static Arrays

- The function $size$ is defined inductively
- Example:
  var a : **array** $[-1..5, 6..7, 3..8]$ **of integer**
  - consists of $5-(-1)+1 = 7$ subarrays of type **array** $[6..7, 3..8]$ **of integer**
  - which consist of $7-6+1 = 2$ subarrays of type **array** $[3..8]$ **of integer**
  - whose sizes are $(8-3+1) \times size(int) = 6$.

  Thus $size(a) = 7 * 2 * 6$ words.
- For an array type

$$at : \mathbf{array}[l_1..u_1, l_2..u_2, \ldots, l_k..u_k] \textbf{ of } t$$

$size(at) = (u_1 - l_1 + 1) * size(\mathbf{array}[l_2..u_2, \ldots, l_k..u_k]\mathbf{of}t)$
Therefore

$$size(at) = \prod_{j=1}^{k} d_j * size(t) \text{ where}$$
$$d_j = u_j - l_j + 1$$

Row Major Array Ordering

var a: **array**$[-5..5, 1..9]$ **of integer**

$$\begin{array}{llll} a[-5,1], & a[-5,2], & \ldots & ,a[-5,9], \\ a[-4,1], & a[-4,2], & \ldots & ,a[-4,9], \\ \vdots & & & \\ a[5,1], & a[5,2], & \ldots & ,a[5,9], \end{array}$$

$$\begin{array}{lcl} \text{l-val}(a[i,j]) & = & \text{l-val}(a) \\ & + & \text{pos. in 1st dimen.} * \text{size of 1-dim. subarray} \\ & + & \text{pos. in 2nd dimen.} * \text{size of int} \\ & = & \text{l-val}(a) \\ & + & (i-(-5)) * (9-1+1) \\ & + & j-1 \\ & = & \text{l-val}(a) \\ & + & (i+5) * 9 + j - 1 \\ & = & \text{l-val}(a) \\ & + & 9 * i + j + 44 \end{array}$$

Row Major General Array Ordering

var a: **array**$[l_1..u_1, l_2..u_2, \ldots, l_n..u_n]$ **of** t

$$\begin{array}{lll} a[l_1,\ldots,l_n], & \ldots & ,a[l_1,\ldots,l_{n-1},u_n], \\ \quad\vdots & & \\ a[u_1,\ldots,u_{n-1},l_n], & \ldots & ,a[u_1,\ldots,u_{n-1},u_n] \end{array}$$

Indexing Static Arrays

Colors: dynamic – static

l-val$(a[\hat{i}_1, \ldots, \hat{i}_n])$

$$\begin{aligned}
= \;& \text{l-val}(a) + \\
& (\hat{i}_1 - l_1) * size(\textbf{array}[l_2..u_2, \ldots, l_n..u_n] \textbf{ of } t) + \\
& (\hat{i}_2 - l_2) * size(\textbf{array}[l_3..u_3, \ldots, l_n..u_n] \textbf{ of } t) + \cdots + \\
& (\hat{i}_n - l_n) * size(t) \\
= \;& \text{l-val}(a) + \\
& (\hat{i}_1 - l_1) * (\textstyle\prod_{j=2}^{n} d_j) * size(t) + \\
& (\hat{i}_2 - l_2) * (\textstyle\prod_{j=3}^{n} d_j) * size(t) + \cdots + \\
& (\hat{i}_n - l_n) * size(t) \\
= \;& \text{l-val}(a) + \\
& \hat{i}_1 * (\textstyle\prod_{j=2}^{n} d_j) * size(t) + \\
& \hat{i}_2 * (\textstyle\prod_{j=3}^{n} d_j) * size(t) + \cdots + \\
& \hat{i}_n * size(t) - \\
& (l_1 * \textstyle\prod_{j=2}^{n} d_j + l_2 * \prod_{j=3}^{n} d_j + l_n) * size(t) \\
= \;& \text{l-val}(a) + \\
& \hat{i}_1 * g * d^{(1)} + \hat{i}_2 * g * d^{(2)} + \cdots + \hat{i}_n * g - d * g
\end{aligned}$$

where $\hat{i}_j =$ r-val(i_j), $\quad d^{(i)} = \prod_{j=i+1}^{n}, \quad g = size(t)$.

P-Code for Array Indexing

Instr.	Meaning	Cond.	Result
ixa q	$STORE[SP-1] :=$ $STORE[SP-1] +$ $STORE[SP] * q;$ $SP := SP - 1$	$\begin{pmatrix} i \\ a \end{pmatrix}$	(a)
inc $T\,q$	$STORE[SP] :=$ $STORE[SP] + q$	$(T),\ Typ(q) = i$	(T)
dec $T\,q$	$STORE[SP] :=$ $STORE[SP] - q$	$(T),\ Typ(q) = i$	(T)

$$\begin{aligned}
code_L\ b[i_1, \ldots, i_n]\ g\ \rho \;&=\; \textbf{ldc a } \rho(b);\ code_I\ [i_1, \ldots, i_n]\ g\ \rho \\
code_I\ [i_1, \ldots, i_n]\ g\ \rho \;&=\; code_R\ i_1\ \rho;\ \textbf{ixa}\ g * d^{(1)}; \\
& \quad\; code_R\ i_2\ \rho;\ \textbf{ixa}\ g * d^{(2)}; \\
& \quad\; \vdots \\
& \quad\; code_R\ i_n\ \rho;\ \textbf{ixa}\ g; \\
& \quad\; \textbf{dec a } g * d;
\end{aligned}$$

Approaches to Array Bound Checking

- No checking at run-time (C)
- Generate a run-time check that the overall index expression is in range (PL1)
- Generate a run-time check that every array index is within range (Pascal, Java)

P-code for Array Checking

Instr.	Meaning	Cond.	Result
chk $p\ q$	**if not** $(p \leq STORE[SP] \leq q)$ **then** $error(\text{“value out of range”})$ **fi**	(i)	(i)

New code for indexing including array bound checks

```
$code_I\ [i_1, \ldots, i_n]\ desc\ \rho =$
    $code_R\ i_1\ \rho;$ **chk** $l_1\ u_1;$ **ixa** $g \cdot d^{(1)};$
    $code_R\ i_2\ \rho;$ **chk** $l_2\ u_2;$ **ixa** $g \cdot d^{(2)};$
      ⋮
    $code_R\ i_n\ \rho;$ **chk** $l_n\ u_n;$ **ixa** $g;$
    **dec** a $g \cdot d;$
```

where $desc = (g; l_1, u_1, \ldots, l_n, u_n)$

This array description is made available through the symbol table, c.f. Semantic Analysis.

Example

var i, j: **integer**;
a: **array**$[-5..5, 1..9]$ **of integer**

$code(a[i+1, j] := 0)\ \rho =$

```
ldc a 7
ldc a 5
ind i
ldc i 1
add i
chk −5 5
ixa 9
ldc a 6
ind i
chk 1 9
ixa 1
dec a −44
ldc i 0
sto i
```

Dynamic Arrays

- The size of the array is determined when the procedure is entered
- Compile-time information:
  - Dimension, i.e. the number of indices
  - Size of the components (if they are not and don't contain dynamic arrays)
- Run-time information:
  - Array bounds
  - Index ranges
  - Size of the array
  - Relative address where the array begins
  - Value of index expressions

Indexing Dynamic Arrays

Colors now: known at indexing – known at creation

l-val$(a[\hat{i}_1, \ldots, \hat{i}_n])$

$$
\begin{aligned}
= \; & \text{l-val}(a) + \\
& (\hat{i}_1 - l_1) * (\textstyle\prod_{j=2}^{n} d_j) * size(t) + \\
& (\hat{i}_2 - l_2) * (\textstyle\prod_{j=3}^{n} d_j) * size(t) + \cdots + \\
& (\hat{i}_n - l_n) * size(t) \\
= \; & \text{l-val}(a) + \\
& (\cdots((\hat{i}_1 * d_2 + \hat{i}_2) * d_3 + \cdots + \hat{i}_n) * d_n * size(t) - \\
& (\cdots((l_1 * d_2 + l_2) * d_3 + \cdots + l_n) * d_n * size(t) \\
= \; & \text{l-val}(a) + d_a - d_c
\end{aligned}
$$

where $\hat{i}_j$ is an abbreviation for $rval(i_j)$.
$d_j = u_j - l_j + 1$ and d_c are computed once, upon array creation.
d_a is computed for each array access.
Computing d_a by a Horner scheme saves multiplications.
Subtracting d_c from the array address yields the "adjusted address".

Array Descriptor

0	Adjusted Address :a
1	Array size :i
2	Subtr. part :i
3	l_1 :i
4	u_1 :i
$\vdots$	$\vdots$
$2n+1$	l_n :i
$2n+2$	u_n :i
$2n+3$	d_2 :i
$\vdots$	$\vdots$
$3n+1$	d_n :i

$code_{Ld}\ c[i_1, \ldots, i_k]\ \rho =$ **ldc** a $\rho(c)$; $code_{Id}\ [i_1, \ldots, i_k]\ g\ \rho$
$code_{Id}\ [i_1, \ldots, i_n]\ g\ \rho\ =$

```
dpl i;                                   descriptor address
ind i;                                   adjusted address
ldc i 0;
code_R $i_1$ $\rho$; add i; ldd $2n+3$; mul i;
code_R $i_2$ $\rho$; add i; ldd $2n+4$; mul i;
  . . .
code_R $i_{n-1}$ $\rho$; add i; ldd $3n+1$; mul i;
code_R $i_n$ $\rho$; add i;
ixa $g$;
sli a
```

Additional P-code instructions

Instr.	Meaning	Cond.	Result
dpl T	$SP := SP + 1;$ $STORE[SP] := STORE[SP-1]$	T	$\begin{pmatrix} T \\ T \end{pmatrix}$
ldd q	$SP := SP + 1;$ $STORE[SP] := STORE[STORE[SP-3]+q]$	$\begin{pmatrix} T_2 \\ T_1 \\ a \end{pmatrix}$	$\begin{pmatrix} i \\ T_2 \\ T_1 \\ a \end{pmatrix}$
sli T_2	$STORE[SP-1] := STORE[SP];$ $SP := SP - 1$	$\begin{pmatrix} T_2 \\ T_1 \end{pmatrix}$	(T_1)

A Simplified Example

program test;
var i : **integer** ; /* $\rho(i) = 5$ */
procedure p ;
var j, a[1..5, 1..i] **of integer** ; /* $\rho(j) = 5, \rho(a) = 6$ */
begin a[3, j] := $\cdots$ /*

```
                ldc a 6
                dpl i
                ind i
                ldc i 0
                ldc i 3
                add i
                ldd 2 * 2 + 3
                mul i
                ldc a 5
                ind i
                add i
                ixa 1
                sli a
                . . .
                sto i */
```

end;
begin read(i); p **end.**

Records

- Size known at compile time
- We assume “unique names”
- Example:

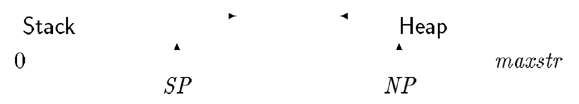
var v : **record** /* $\rho(v) = 5$*/
    a : **integer** ; /* $\rho(a) = 0$ */
    b : **integer** /* $\rho(b) = 1$ */
end

$code_L$(v.b) = **ldc** a 5 ; **inc** a 1

Pointers and dynamic memory allocation

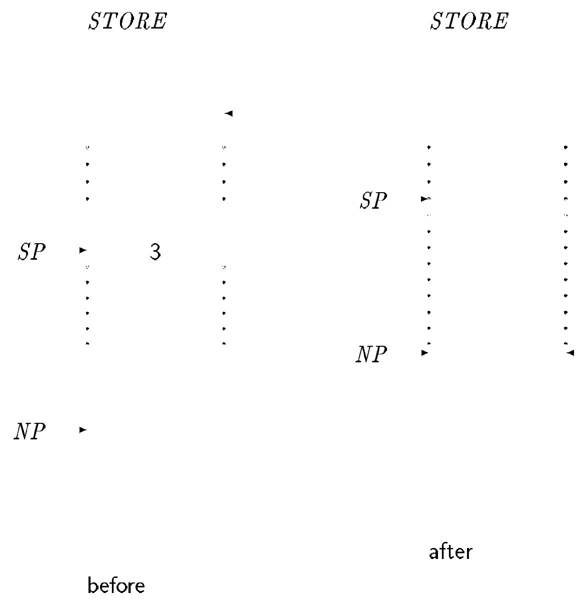
- Dynamic duration (lifetime) of the memory
- Storage can be freed in one of the following cases:
  - Can no longer be refererenced (“garbage collection”)
  - Explicit deallocation (**cfree**, **dispose**)
- Space cannot be allocated on the “stack”

The Heap of the P-Machine



Instruct.	Meaning	Cond.	Result
new	**if** $NP - STORE[SP] \leq EP$ **then** *error* ("store overflow") **else** $NP := NP - STORE[SP]$; $STORE[STORE[SP-1]] := NP$; $SP := SP - 2$ **fi**;	$\binom{i}{a}$	

The Effect of New Statement



The Generated P-Code

Assume that $Typ(x) = \uparrow t$

$code(\textbf{new}(x))\ \rho\ =$

ldc a $\rho(x)$
ldc i $size(t)$
new

Complex Pointer Expressions

- Examples:
  - $x \uparrow [i+1, j].a \uparrow [i] \uparrow$
  - $y.a.b.c \uparrow [i, j+1].d$
- Recursive translation

$$\begin{array}{lcll}
code_L(xr)\ \rho & = & \textbf{ldc}\ \mathrm{a}\ \rho(x); & \text{for name } x \\
 & & code_M(r)\ \rho & \\
code_M(.xr)\ \rho & = & \textbf{inc}\ \mathrm{a}\ \rho(x); & \text{for name } x \\
 & & code_M(r)\ \rho & \\
code_M(\uparrow r)\ \rho & = & \textbf{ind}\ \mathrm{a}; & \\
 & & code_M(r)\ \rho & \\
code_M([i_1, \ldots, i_n]r)\ \rho & = & code_{Id}\ [i_1, \ldots, i_n]\ g\ \rho; & \\
 & & code_M(r)\ \rho & \text{for array} \\
code_M(\varepsilon)\ \rho & = & \varepsilon &
\end{array}$$

Example

type	$t =$	**record**
		a : **array**$[-5..+5, 1..9]$ **of integer**;
		$b :\uparrow t$
		end;
var	i, j	: **integer**;
	pt	$:\uparrow t$;

$code_L\ pt \uparrow .b \uparrow .a[i+1, j]$

ldc a $\rho(pt)$;	Load Address of pt
ind a;	Load start address of the record
inc a 99;	The start address of b
ind a;	Dereference pointer
inc a 0;	The start address of a
$code_{Id}[i+1, j]\ 1\ \rho$	

Handling Procedures

Mooly Sagiv
Tel Aviv University
sagiv@math.tau.ac.il
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

April 29, 1997

Subjects

- Scoping
- Static vs. Dynamic Binding
- Calling Trees
- Static Predecessor Tree
- Parameter Passing Mechanisms
- The Run-Time Stack Frame
- Addressing of Variables
- Computing the Address Environment
- Procedures as Parameters

Procedure Incarnations

- Calling a procedure p creates an *incarnation* $\hat{p}$
- An incarnation of a procedure contains incarnations of the formal parameters and the local variables
- Control

1. enters the incarnation $\hat{p}$ from the caller $\hat{q}$,
2. may pass to a procedure incarnation $\hat{r}$ created by a call,
3. returns from $\hat{r}$,
4. returns from $\hat{p}$ to $\hat{q}$.

Calling Tree

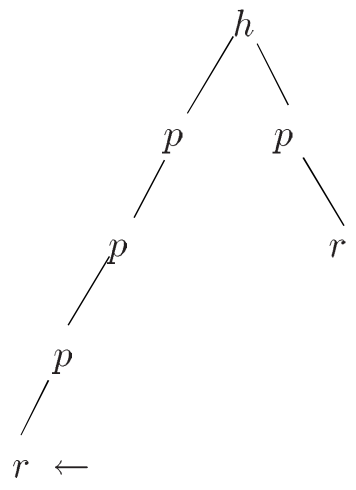
- Defined for an execution sequence
- An ordered tree
- The root is the main program
- There is a node for every procedure incarnation
- If p **calls** $p_1, p_2, \ldots, p_n$, then incarnations of the $p_1, p_2, \ldots, p_n$ are the children of p

A path starting with the root is called an **incarnation path**.

Example: Calling Tree

```
program $h$;
    var $i$ : integer;
    proc $p$
        proc $r$

        if $i > 0$
            then $i := i - 1$; $p$
            else $r$
        fi
    $i := 2$;
    $p$;
    $p$
end.
```



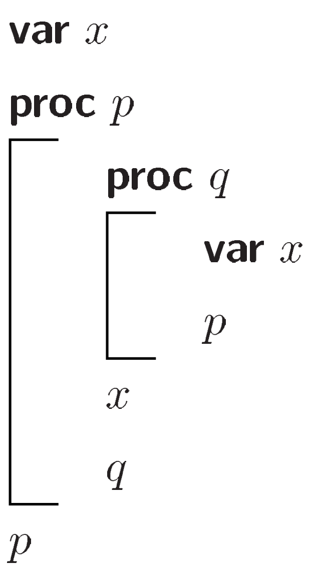
A program and its calling tree

Static vs. Dynamic Binding

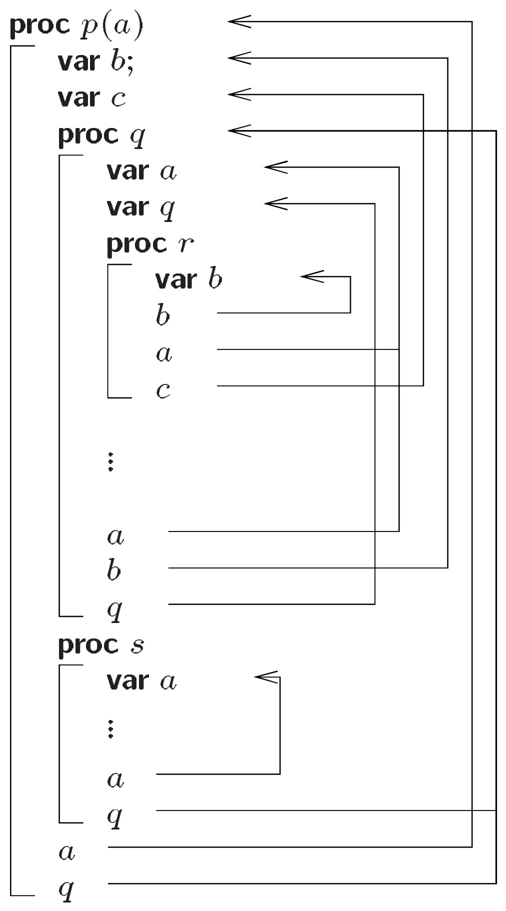
Binding strategy (*scope rules*) relates *applied* occurrences of names to *defining* occurrences.

Static: Applied occurrence of x denotes the defining occurrence of x in the next enclosing scope,

Dynamic: Applied occurrence of x denotes the last created incarnation of x.



Static Binding



Static Predecessor Tree

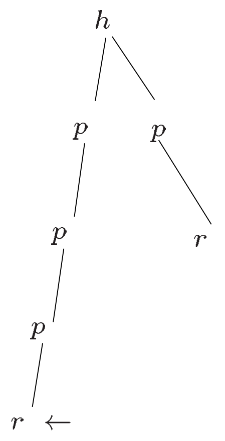
- Defined for an incarnation path of a call tree
- Its root is the main program
- There is a node for every procedure incarnation of the path
- $\hat{p}$ is parent of $\hat{q}$, iff q is declared directly inside p and $\hat{p}$ is the first incarnation of p "above" $\hat{q}$ in the path

Warning! In general only true for programs without formal procedures.

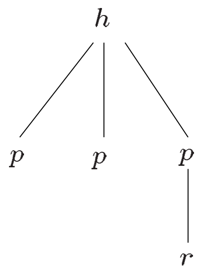
Example: Static Predecessor Trees

```
program h;
    var i : integer;
    proc p
        proc r


        if i > 0
            then i := i − 1; p
            else r
        fi
    i := 2;
    p;
    p
end.
```



A program and its calling tree



Its static predecessor tree

Parameter Passing Mechanisms

value: The r-value of the actual parameter is passed.

value-result: Only variable actual parameters allowed. Their r-value is passed to and returned from the callee.

var/reference: Only variable actual parameters allowed.
Their l-value is passed to the callee.

name: Actual parameter is evaluated every time execution needs it.
"Thunk"" (pointer to evaluation code and pointer to caller's frame) is passed.

Example

```
program test ;
var i : integer ;
procedure p(. . . x, y: integer);
begin
        while $x > 0$ do
                $x := x - 1$ ;
                $y := y + 1$
        end
end
begin
        $i := 3$ ;
        $p(i, i)$
end
```

. . . = value:

. . . = var:

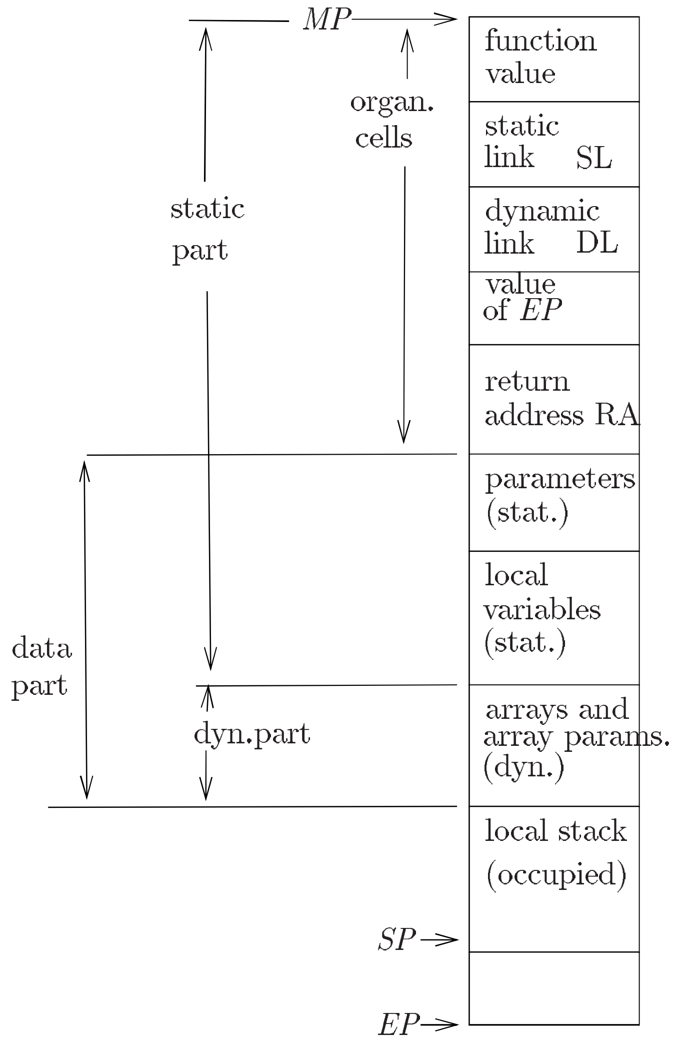
. . . = value-result:

Implementing Recursion

- Traversing the call tree with a nonrecursive program,
- needs a stack,
- elements in the stack are nodes of the call tree (i.e. procedure incarnations),
- stack content is prefix of an incarnation path (starting at the root),
- structure of this stack is isomorphic to the *run-time stack* for implementing procedures.

The Structure of an Activation Record

The representation of a procedure incarnation



Further outline

- Conceptual view of addressing
- Adapt memory allocation scheme ρ
- Adapt code for l-values
- Code for function/procedure calls/return
- Code for parameter passing
- Code for the function/procedure body

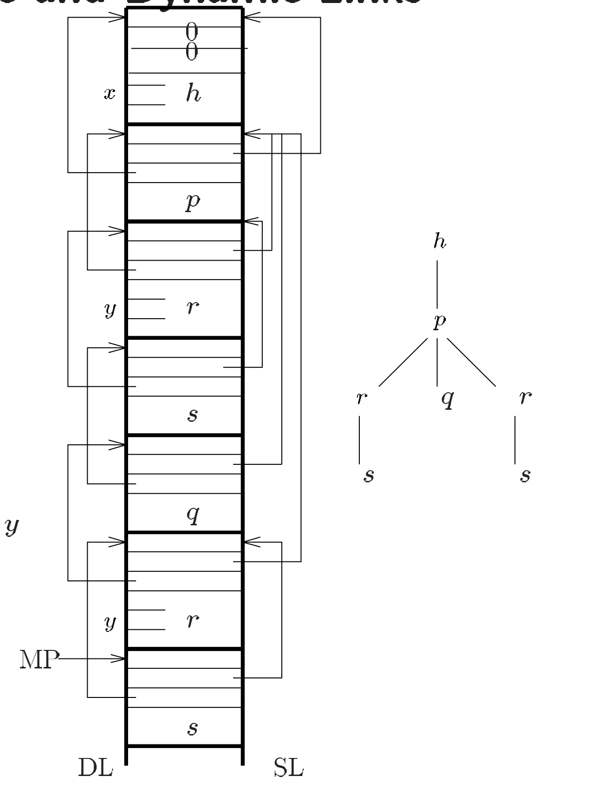
Addressing Global Variables

Using *nesting depth* of name occurrences
Inductively defined

- $nd(main) = 0$
- Types, variables, and procedures declared inside procedure p have nesting depth $nd(p) + 1$
- Applied occurrences of names inside procedure p have nesting depth $nd(p) + 1$

Static and Dynamic Links

```
program h;
  var x
  proc p
    ⋮
    proc q
      ⋮
      r
      ⋮
    proc r
      var y
      proc s
        x + y
        q
      s
    r
  p
```

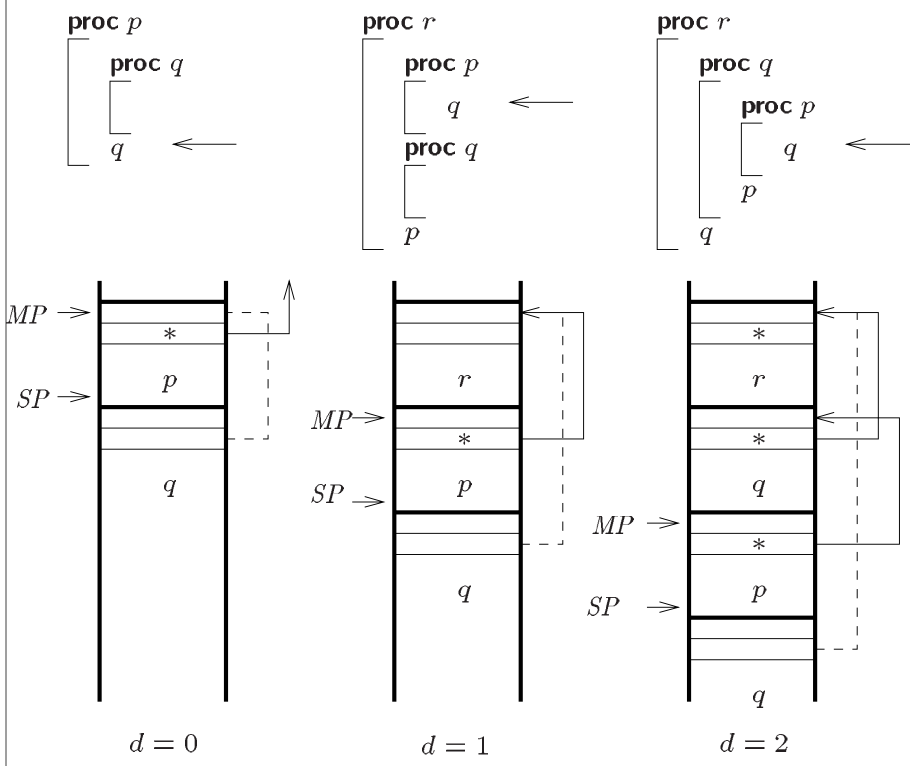


program stack configuration corresp. tree of static pred.

Defining	occurrence	Applied	occurrence
p	1	p	1
q	2	q	4
r	2	r (in p)	2
s	3	r (in q)	3

Nesting depths

Updating the Static Link



Three different calls and stack configurations.

Adapt Memory Allocation Scheme

- $\rho(x) = \langle \textit{relative-address}, \textit{nesting depth} \rangle$
- Setting relative address remains:

$$\rho_1(v_i) = 5 + \sum_{j=0}^{i-1} size(t_j)$$

for **var** $v_0 : t_0; v_1 : t_2; \ldots ; v_k : t_k;$

Size Setting

Static sizes:

Parameters

var-parameters $size = 1$
value-dyn-array-parameters $size = 3 * n + 2$
other-value-parameters $size$ is unchanged

Locals

static-array $size = 3 * n + 2 +$ old-size
other-local $size$ is unchanged

$3 * n + 2$ size of array descriptor

Dynamic sizes:

Dynamic arrays (locals+parameters)
as given by old $size$ formula, but evaluated at run-time.

Adapt $code_L$

Instr.	Meaning
lod $T\ p\ q$	$SP := SP + 1;$
	$STORE[SP] := STORE[base(p, MP) + q]$
lda $p\ q$	$SP := SP + 1;$
	$STORE[SP] := base(p, MP) + q$
str $T\ p\ q$	$STORE[base(p, MP) + q] := STORE[SP];$
	$SP := SP - 1$

$base(p, a)$ = **if** $p = 0$ **then** a **else** $base(p-1, STORE[a+1])$

$code_L(x\ r)\ \rho\ nd =$ **lda** a d ra;
$code_M\ r\ \rho\ nd$,
where $\rho(x) = (ra, nd')$, $d = nd - nd'$ and
x is variable or formal value parameter

$code_L(x\ r)\ \rho\ nd =$ **lod** a d ra;
$code_M\ r\ \rho\ nd$
where $\rho(x) = (ra, nd')$, $d = nd - nd'$,
and x is formal var parameter

Code for Procedure/Function Call

1. Set static link of callee
2. Set dynamic link of callee
3. Save EP
4. Evaluate parameters (l-values, r-values)
5. Set MP
6. Save PC as return-address
7. Jump to code of callee
8. Set SP to end of static part
9. Allocate space for dynamic value arrays; copy
10. Set EP

Procedure Call

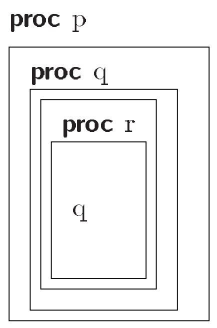
```
$code\ p(e_1, \ldots, e_k)\ \rho\ nd =$
    mst $nd - nd'$;
    $code_A\ e_1\ \rho\ nd$;
        ⋮
    $code_A\ e_k\ \rho\ nd$;
    cup $s\ l$                    (* $\rho(p) = (l, nd)$ *)
```

P-instructions for call and entry

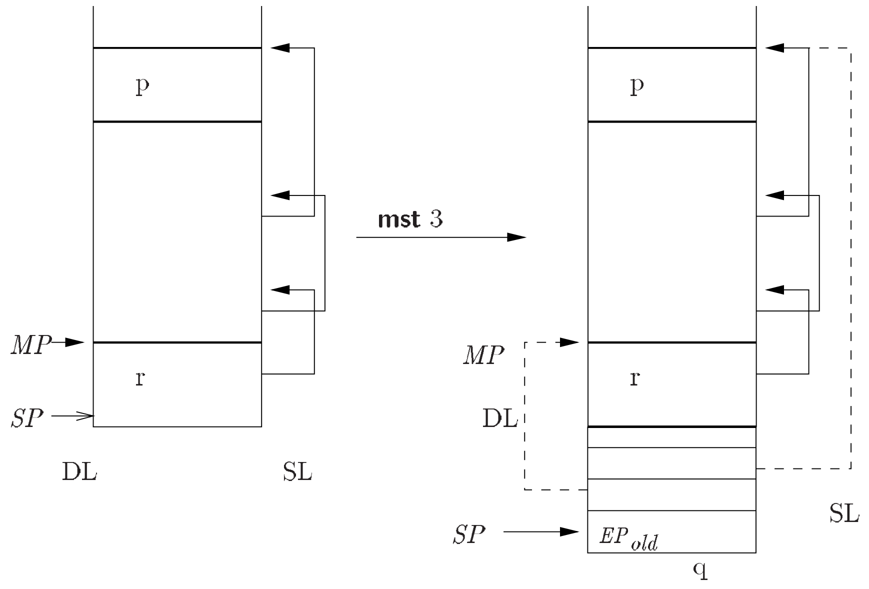
Instr.	Meaning	Comment
mst p	$STORE[SP+2] := base(p, MP)$;	Static link
	$STORE[SP+3] := MP$;	Dynamic link
	$STORE[SP+4] := EP$;	Save EP
	$SP := SP + 5$	
cup $p\ q$	$MP := SP - (p+4)$;	p space
		for parameters
	$STORE[MP+4] := PC$;	Return address
	$PC := q$	Jump to q
ssp p	$SP := MP + p - 1$	alloc. static area
sep p	$EP := SP + p$;	alloc. temp. area
	if $EP \geq NP$	Check collision
	then *error* (*“store overflow”*)	stack and heap
	fi	

$base(p, a)$ = **if** $p = 0$ **then** a **else** $base(p-1, STORE[a+1])$**fi**

P-instruction mst



nd(applied occurrence of q) -
nd(defining occurrence of q) = 3



Code for Procedure/Function Return

1. Restore SP to the beginning of current stack frame
2. Restore PC to return-address
3. Restore EP and check for heap-stack collision
4. Release frame, i.e. set MP to dynamic link

P-instructions for Return

Instr.	Meaning	Comment
retf	$SP := MP$;	result is on top
	$PC := STORE[MP+4]$;	return address
	$EP := STORE[MP+3]$;	Restore EP
	if $EP \geq NP$	
	then *error*(*"store overflow"*)	
	fi	
	$MP := STORE[MP+2]$	Release frame
retp	$SP := MP - 1$;	No return value
	$PC := STORE[MP+4]$;	return address
	$EP := STORE[MP+3]$;	Restore EP
	if $EP \geq NP$	
	then *error*(*"store overflow"*)	
	fi	
	$MP := STORE[MP+2]$	Release frame

Procedure/Function Code

```
code (procedure p (specs); vdecls; pdecls; body) ρ nd =
        ssp n_a'';
        code_P specs ρ' nd;
        code_P vdecls ρ'' nd;
        sep  k;
        ujp  l;
        proc_code;                       (* local procedures *)
     l : code body ρ''' nd;
        retp/retf
     where     (ρ', n_a') = elab_specs specs ρ 5 nd
               (ρ'', n_a'') = elab_vdecls vdecls ρ' n_a' nd
               (ρ''', proc_code) = elab_pdecls pdecls ρ'' nd
```

$elab\_specs$

$$elab\_specs : \; Spec^* \times \; Addr\_Env \; \times \; Addr \; \times \; ND \; \rightarrow \; Addr\_Env \; \times \; Addr$$

$elab\_specs$ (**var** $x : t;$ $specs$) ρ n_a $nd =$
$\quad elab\_specs$ $specs$ $\rho[\,(n\_a, nd)/x\,]$ $(n\_a + 1)$ nd

$elab\_specs$ (**value** x: **array**$[l_1..u_1, \ldots, l_k..u_k]$ **of** t'; $specs$)
$\qquad \rho$ n_a $nd =$
$\quad elab\_specs$ $specs$ ρ' $(n\_a + 3k + 2)$ nd $where$
$\qquad \rho' = \rho[(n\_a, nd)/x]$
$\qquad\qquad [(n\_a + 2i + 1, nd)/l_i]_{i=1}^{k}$
$\qquad\qquad [(n\_a + 2i + 2, nd)/u_i]_{i=1}^{k}$

$elab\_specs$ (**value** $x : t;$ $specs$) ρ n_a $nd =$
$\quad elab\_specs$ $specs$ $\rho[\,(n\_a, nd)/x\,]$ $(n\_a + size(t))$ nd
$\qquad\qquad$ for static type t

$elab\_specs$ ( ) ρ n_a $nd = (\rho, n\_a)$

$elab\_pdecls$

$elab\_pdecls$ processes procedure declarations

$$elab\_pdecls : Pdecl^* \times \; Addr\_Env \; \times \; ND \rightarrow Addr\_Env \; \times \; Code$$

$elab\_pdecls$ (**proc** $p_1(\ldots); \ldots;$
$\qquad\qquad \vdots$
$\qquad\quad$ **proc** $p_k(\ldots); \ldots;$) ρ $nd =$

$\qquad (\rho', \qquad l_1 : code$ (**proc** $p_1(\ldots); \ldots)$ ρ' $nd + 1;$
$\qquad\qquad\qquad \vdots$
$\qquad\qquad\quad l_k : code$ (**proc** $p_k(\ldots); \ldots)$ ρ' $nd + 1)$
where $\rho' = \rho[(l_1, nd)/p_1, \ldots, (l_k, nd)/p_k]$

$elab\_pdecls$ ( ) ρ $nd = (\rho, (\;))$

Parameter Passing

var-actual-parameters

$$code_A\ x\ \rho\ nd = \\ \quad code_L\ x\ \rho\ nd$$

value-actual-parameters

$$code_A\ e\ \rho\ nd = \\ \quad code_R\ e\ \rho\ nd$$

value-actual-structural-parameters

$$code_A\ x\ \rho\ nd = \\ \quad code_L\ x\ \rho\ nd; \\ \quad \textbf{movs}\ g$$

P-code for moves

Instr.	Meaning	Cond.	Res.
movs q	**for** $i := q-1$ **down to** 0 **do** $STORE[SP+i] :=$ $STORE[STORE[SP]+i]$ **od**; $SP := SP + q - 1$	(a)	
movd q	**for** $i := 1$ **to** $STORE[MP+q+1]$ **do** $STORE[SP+i] :=$ $STORE[STORE[MP+q]$ $+STORE[MP+q+2]+i-1]$ **od**; $STORE[MP+q] :=$ $SP+1-STORE[MP+q+2]$ $SP := SP + STORE[MP+q+1]$		

Copying Dynamic Arrays

$code_P$ (**value** x : **array**$[u_1..o_1,\ldots,u_k..o_k]$ **of** t) $\rho\ nd =$
  movd ra;

The Main Program

```
$code$ (**program** $p$ ($specs$); $vdecls$; $pdecls$; $body$) $\rho$ $0 =$
        **ssp** $n\_a$;
        $code_P$ $vdecls$ $\rho$ $1$;
        **sep** $k$;
        **ujp** $l$;
        $proc\_code$;
    $l$ : $code$ $body$ $\rho'$ $1$;
        **stp**
where $(\rho, n\_a) = elab\_vdecls\ vdecls\ \emptyset\ 5\ 1$  and
          $(\rho', proc\_code) = elab\_pdecls\ pdecls\ \rho\ 1$
```

Input Example

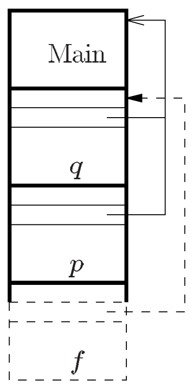
```
program foo ;
var i : integer ;
ssp 10; sep 7; ujp $l_1$;
function fact( n: integer): integer;
$l_2$: ssp 6; sep 9; ujp $l_3$;
begin $l_3$ :
      if $n = 1$
        lda a 2 − 2 5; ind $i$; ldc $i$ 1; equ $i$; fjp $l_4$
        then $fact := 1$
        lda a 2 − 2 0; ldc $i$ 1; sto $i$; ujp $l_5$ :
        else $fact := n * fact(n - 1)$ fi
        $l_4$: lda a 2 − 2 0; lda a 2 − 2 5; ind $i$;
        mst 2 − 1; lda a 2 − 2 5; ind $i$; ldc $i$ 1; sub $i$; cup 1 $l_2$
        mul $i$; sto $i$; $l_5$:
end; retf
begin $l_1$ :
      $i := fact(2)$
      lda a 1 − 1 9;
      mst 1 − 1; ldc $i$ 2; cup 1 $l_2$
      sto $i$
end. stp
```

Procedures/Functions as Parameters

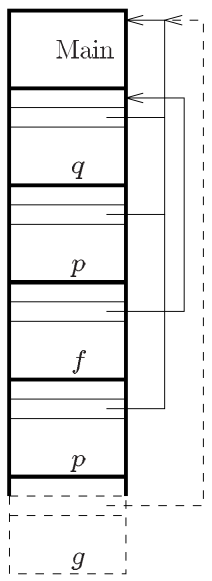
```
program      Main;
    proc p( function h)
        h
    proc q
        function f
            p(g)
        p(f)
    function g

    q
```

Program with formal procedures



Stacksituation after call $p(f)$ and (dashed) after call h



Stacksituation after call $p(g)$ and (dashed) after call h

EECS 583 – Lecture 15 Machine Information, Scheduling a Basic Block

University of Michigan

March 5, 2003

Machine Information

- Each step of code generation requires knowledge of the machine
  - Hard code it? – used to be common practice
  - Retargetability, then cannot
- What does the code generator need to know about the target processor?
  - Structural information?
    - No
  - For each opcode
    - What registers can be accessed as each of its operands
    - Other operand encoding limitations
  - Operation latencies
    - Read inputs, write outputs
  - Resources utilized
    - Which ones, when

Machine Description (mdes)

- Elcor mdes supports very general class of EPIC processors
  - Probably more general than you need ☺
  - Weakness – Does not support ISA changes like GCC
- Terminology
  - Generic opcode
    - Virtual opcode, machine supports k versions of it
    - ADD_W
  - Architecture opcode or unit specific opcode or sched opcode
    - Specific assembly operation of the processor
    - ADD_W.0 = add on function unit 0
- Each unit specific opcode has 3 properties
  - IO format
  - Latency
  - Resource usage

IO Format

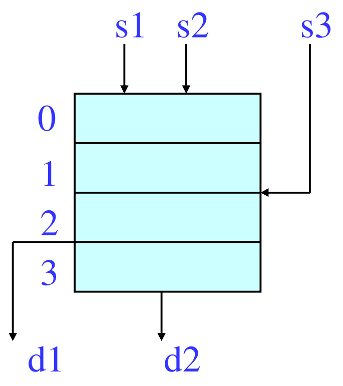
- Registers, register files
  - Number, width, static or rotating
  - Read-only (hardwired 0) or read-write
- Operation
  - Number of source/dests
  - Predicated or not
  - For each source/dest/pred
    - What register file(s) can be read/written
    - Literals, if so, how big

Multicluster machine example:

ADD_W.0	gpr1, gpr1 : gpr1
ADD_W_L.0	gpr1, lit6 : gpr1
ADD_W.1	gpr2, gpr2 : gpr2

Latency Information

- Multiply takes 3 cycles
  - » No, not that simple!!!
- Differential input/output latencies
  - » Earliest read latency for each source operand
  - » Latest read latency for each source operand
  - » Earliest write latency for each destination operand
  - » Latest write latency for each destination operand
- Why all this?
  - » Unexpected events may make operands arrive late or be produced early
- Compound op: part may finish early or start late
- Instruction re-execution by
  - » Exception handlers
  - » Interupt handlers
- Ex: mpyadd(d1, d2, s1, s2, s3)
  - » d1 = s1 * s2, d2 = d1 + s3



E/L
s1: 0/2
s2: 0/2
s3: 2/2
d1: 2/3
d2: 2/4

Memory Serialization Latency

- Ensuring the proper ordering of dependent memory operations
- Not the memory latency
  - » But, point in the memory pipeline where 2 ops are guaranteed to be processed in sequential order
- Page fault – memory op is re-executed, so need
  - » Earliest mem serialization latency
  - » Latest mem serialization latency
- Remember
  - » Compiler will use this, so any 2 memory ops that cannot be proven independent, must be separated by mem serialization latency.

Branch Latency

- Time relative to the initiation time of a branch at which the target of the branch is initiated
- What about branch prediction?
  - Can reduce branch latency
  - But, may not make it 1
- We will assume branch latency is 1 for this class (ie no delay slots!)

Example:

0: branch
1: xxx
2: yyy
3: target

branch latency = k (3)
delay slots = k – 1 (2)
Note xxx and yyy are multiOps

Resources

- A machine resource is any aspect of the target processor for which over-subscription is possible if not explicitly managed by the compiler
  - Scheduler must pick conflict free combinations
- 3 kinds of machine resources
  - Hardware resources are hardware entities that would be occupied or used during the execution of an opcode
    - Integer ALUS, pipeline stages, register ports, busses, etc.
  - Abstract resources are conceptual entities that are used to model operation conflicts or sharing constraints that do not directly correspond to any hardware resource
    - Sharing an instruction field
  - Counted resources are identical resources such that k are required to do something
    - Any 2 input busses

Reservation Tables

For each opcode, the resources used at each cycle relative to its initiation time are specified in the form of a table

Res1, Res2 are abstract resources to model issue constraints

relative time	Res1	Res2	ALU	MPY	Resultbus
0	X		X		
1					X

Integer add

relative time	Res1	Res2	ALU	MPY	Resultbus
0		X		X	
1				X	
2					X

Non-pipelined multiply

relative time	Res1	Res2	ALU	MPY	Resultbus
0	X	X	X		
1					X

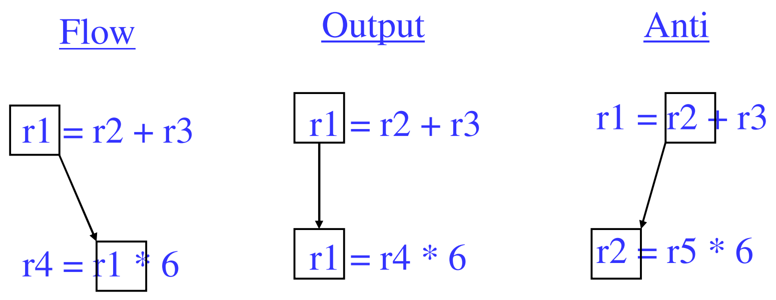
Load, uses ALU for addr calculation, can't issue load with add or multiply

Now, Lets Get Back to Scheduling…

- Scheduling constraints
  - » What limits the operations that can be concurrently executed or reordered?
  - » Processor resources – modeled by mdes
  - » Dependences between operations
    - Data, memory, control
- Processor resources
  - » Manage using resource usage map (RU_map)
  - » When each resource will be used by already scheduled ops
  - » Considering an operation at time t
    - See if each resource in reservation table is free
  - » Schedule an operation at time t
    - Update RU_map by marking resources used by op busy

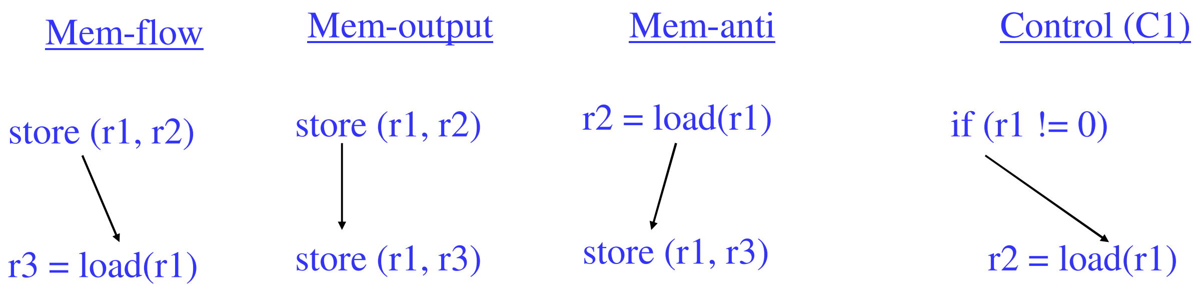
Data Dependences

- Data dependences
  - If 2 operations access the same register, they are dependent
  - However, only keep dependences to most recent producer/consumer as other edges are redundant
  - Types of data dependences



More Dependences

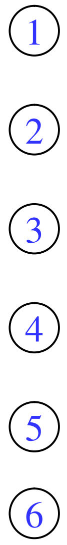
- Memory dependences
  - Similar as register, but through memory
  - Memory dependences may be certain or maybe
- Control dependences
  - We discussed this earlier
  - Branch determines whether an operation is executed or not
  - Operation must execute after/before a branch
  - Note, control flow (C0) is not a dependence



Dependence Graph

- Represent dependences between operations in a block via a DAG
  - » Nodes = operations
  - » Edges = dependences
- Single-pass traversal required to insert dependences
- Example

```
    1: r1 = load(r2)
    2: r2 = r1 + r4
    3: store (r4, r2)
    4: p1 = cmpp (r2 < 0)
    5: branch if p1 to BB3
    6: store (r1, r2)
BB3:
```



Dependence Edge Latencies

- Edge latency = minimum number of cycles necessary between initiation of the predecessor and successor in order to satisfy the dependence
- Register flow dependence, a → b
  - » Latest_write(a) – Earliest_read(b)
- Register anti dependence, a → b
  - » Latest_read(a) – Earliest_write(b) + 1
- Register output dependence, a → b
  - » Latest_write(a) – Earliest_write(b) + 1
- Negative latency
  - » Possible, means successor can start before predecessor
  - » We will only deal with latency >= 0, so MAX any latency with 0

Dependence Edge Latencies (2)

- Memory dependences, a → b (all types, flow, anti, output)
  - latency = latest_serialization_latency(a) – earliest_serialization_latency(b) + 1
  - Prioritized memory operations
    - Hardware orders memory ops by order in MultiOp
    - Latency can be 0 with this support
- Control dependences
  - branch → b
    - Op b cannot issue until prior branch completed
    - latency = branch_latency
  - a → branch
    - Op a must be issued before the branch completes
    - latency = 1 – branch_latency (can be negative)
    - conservative, latency = MAX(0, 1-branch_latency)

Class Problem

machine model

min/max read/write latencies

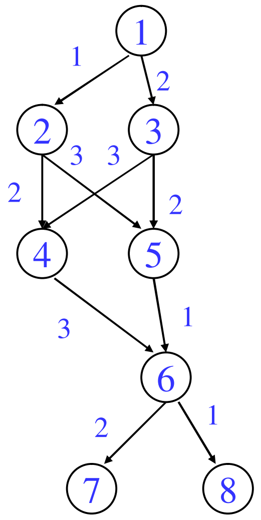
add:	src	0/1
	dst	1/1
mpy:	src	0/2
	dst	2/3
load:	src	0/0
	dst	2/2
	sync	1/1
store:	src	0/0
	dst	-
	sync	1/1

1. Draw dependence graph
2. Label edges with type and latencies

```
r1 = load(r2)
r2 = r2 + 1
store (r8, r2)
r3 = load(r2)
r4 = r1 * r3
r5 = r5 + r4
r2 = r6 + 4
store (r2, r5)
```

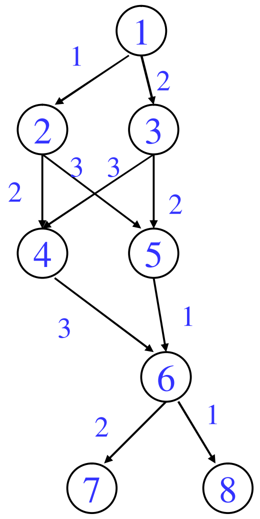
Dependence Graph Properties - Estart

- Estart = earliest start time, (as soon as possible - ASAP)
  - Schedule length with infinite resources (dependence height)
  - Estart = 0 if node has no predecessors
  - Estart = MAX(Estart(pred) + latency) for each predecessor node
  - Example



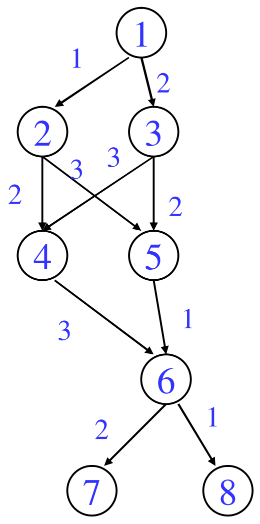
Lstart

- Lstart = latest start time, ALAP
  - Latest time a node can be scheduled s.t. sched length not increased beyond infinite resource schedule length
  - Lstart = Estart if node has no successors
  - Lstart = MIN(Lstart(succ) - latency) for each successor node
  - Example



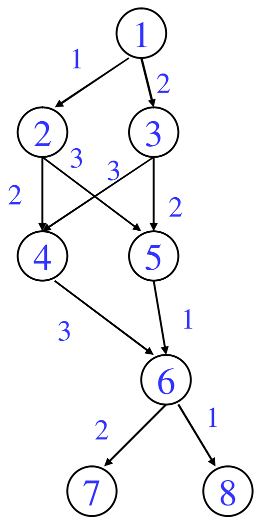
Slack

- Slack = measure of the scheduling freedom
  - Slack = Lstart – Estart for each node
  - Larger slack means more mobility
  - Example

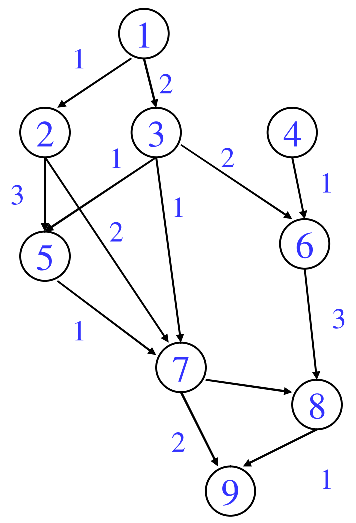


Critical Path

- Critical operations = Operations with slack = 0
  - No mobility, cannot be delayed without extending the schedule length of the block
  - Critical path = sequence of critical operations from node with no predecessors to exit node, can be multiple crit paths



Class Problem



Node	Estart	Lstart	Slack
1			
2			
3			
4			
5			
6			
7			
8			
9			

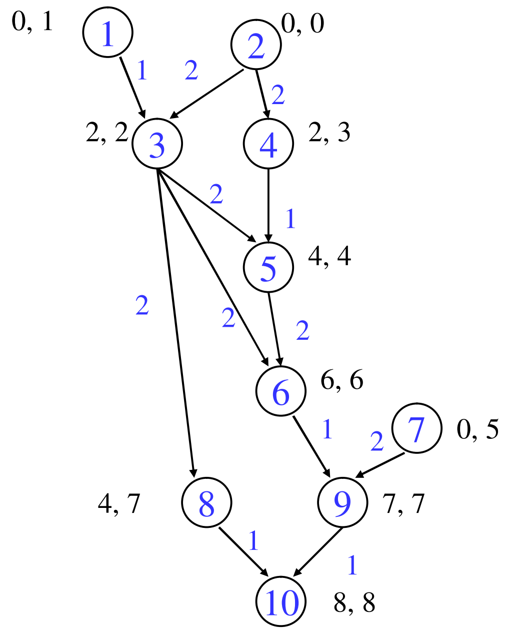
Critical path(s) =

Operation Priority

- Priority – Need a mechanism to decide which ops to schedule first (when you have multiple choices)
- Common priority functions
  - Height – Distance from exit node
    - Give priority to amount of work left to do
  - Slackness – inversely proportional to slack
    - Give priority to ops on the critical path
  - Register use – priority to nodes with more source operands and fewer destination operands
    - Reduces number of live registers
  - Uncover – high priority to nodes with many children
    - Frees up more nodes
  - Original order – when all else fails

Height-Based Priority

- Height-based is the most common
  - priority(op) = MaxLstart – Lstart(op) + 1



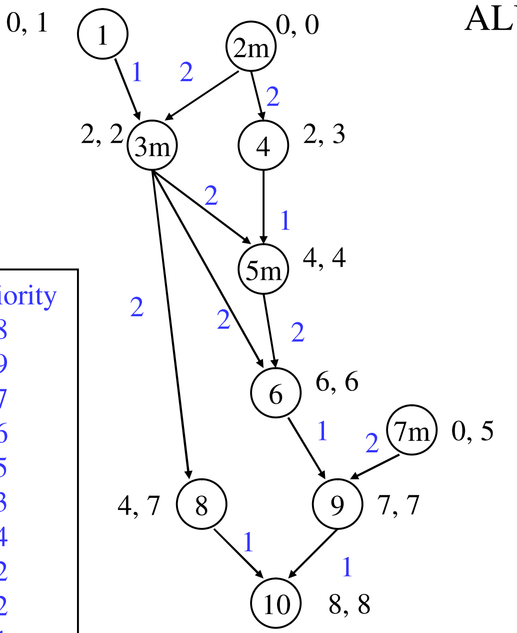
op	priority
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

List Scheduling (Cycle Scheduler)

- Build dependence graph, calculate priority
- Add all ops to UNSCHEDULED set
- time = -1
- while (UNSCHEDULED is not empty)
  - time++
  - READY = UNSCHEDULED ops whose incoming dependences have been satisfied
  - Sort READY using priority function
  - For each op in READY (highest to lowest priority)
    - op can be scheduled at current time? (are the resources free?)
      - Yes, schedule it, op.issue_time = time
        - Mark resources busy in RU_map relative to issue time
        - Remove op from UNSCHEDULED/READY sets
      - No, continue

Cycle Scheduling Example

Machine: 2 issue, 1 memory port, 1 ALU
Memory port = 2 cycles, non-pipelined
ALU = 1 cycle

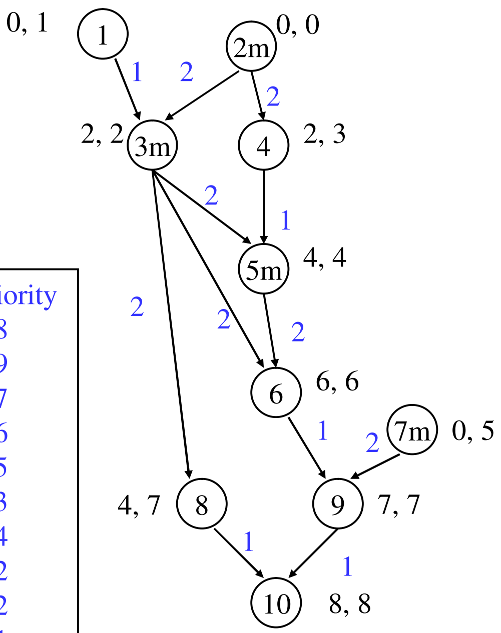


op	priority
1	8
2	9
3	7
4	6
5	5
6	3
7	4
8	2
9	2
10	1

RU_map

time	ALU	MEM
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Cycle Scheduling Example (2)



op	priority
1	8
2	9
3	7
4	6
5	5
6	3
7	4
8	2
9	2
10	1

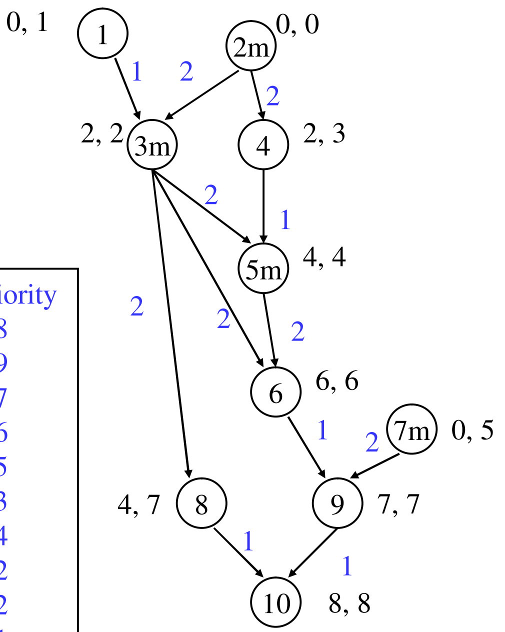
RU_map

time	ALU	MEM
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Schedule

time	Ready	Placed
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Cycle Scheduling Example (3)



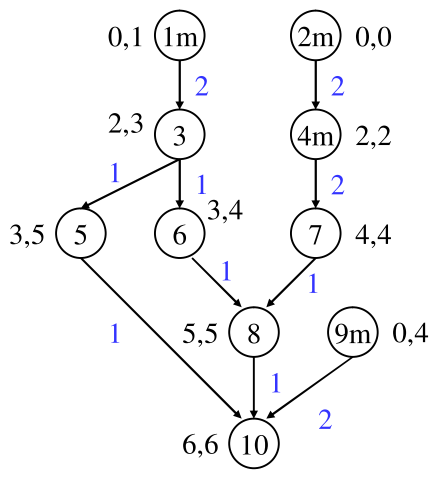
op	priority
1	8
2	9
3	7
4	6
5	5
6	3
7	4
8	2
9	2
10	1

Schedule

time	Ready	Placed
0	1,2,7	1,2
1	7	-
2	3,4,7	3,4
3	7	-
4	5,7,8	5,8
5	7	-
6	6,7	6,7
7	-	
8	9	9
9	10	10

Class Problem

Machine: 2 issue, 1 memory port, 1 ALU
Memory port = 2 cycles, pipelined
ALU = 1 cycle



1. Calculate height-based priorities
2. Schedule using cycle scheduler