Compiler Technology of Programming Languages

Code Generation
Project Implementation Hints

Prof. Farn Wang

Taking the first step

```
int main() {
   int lid;
   lid = 1;
}
```

Function declaration

Prologue

Variable declaration

Assignment statement

Epilogue

Question:

what are the minimal code to insert in order to handle this elementary routine

Example: ARM64

```
int gid;
int main() {
   gid = 1;
}
```

```
.data
_gid:    .word 0
.text
main:
        str x30, [sp, #0]    return address
        str x29, [sp, #-8]   old fp
        add x29, sp, #-8
        sp, #-16
        =_frameSize_MAIN
        30, #0]
        w30    push AR
        #8]    save states
        p, #16]

        mov w9, #1
        ldr x10, =_g_gid
        str w9, [x10, #0]
_end_main:
        Epilogue
data
frameSize_MAIN: .word 8
```

Note, this is only AARCH64 convention! Your project is not bound to it.

```
ldr x9, [sp, #8]
ldr x10, [sp, #16]
.....
ldr x30, [x29, #8]
mov sp, x29
add sp, sp, #8
ldr x29, [x29,#0]
RET x30
```

```
int gid;
int main() {
   gid = 1;
}
```

Why not wait until Declaration is done?

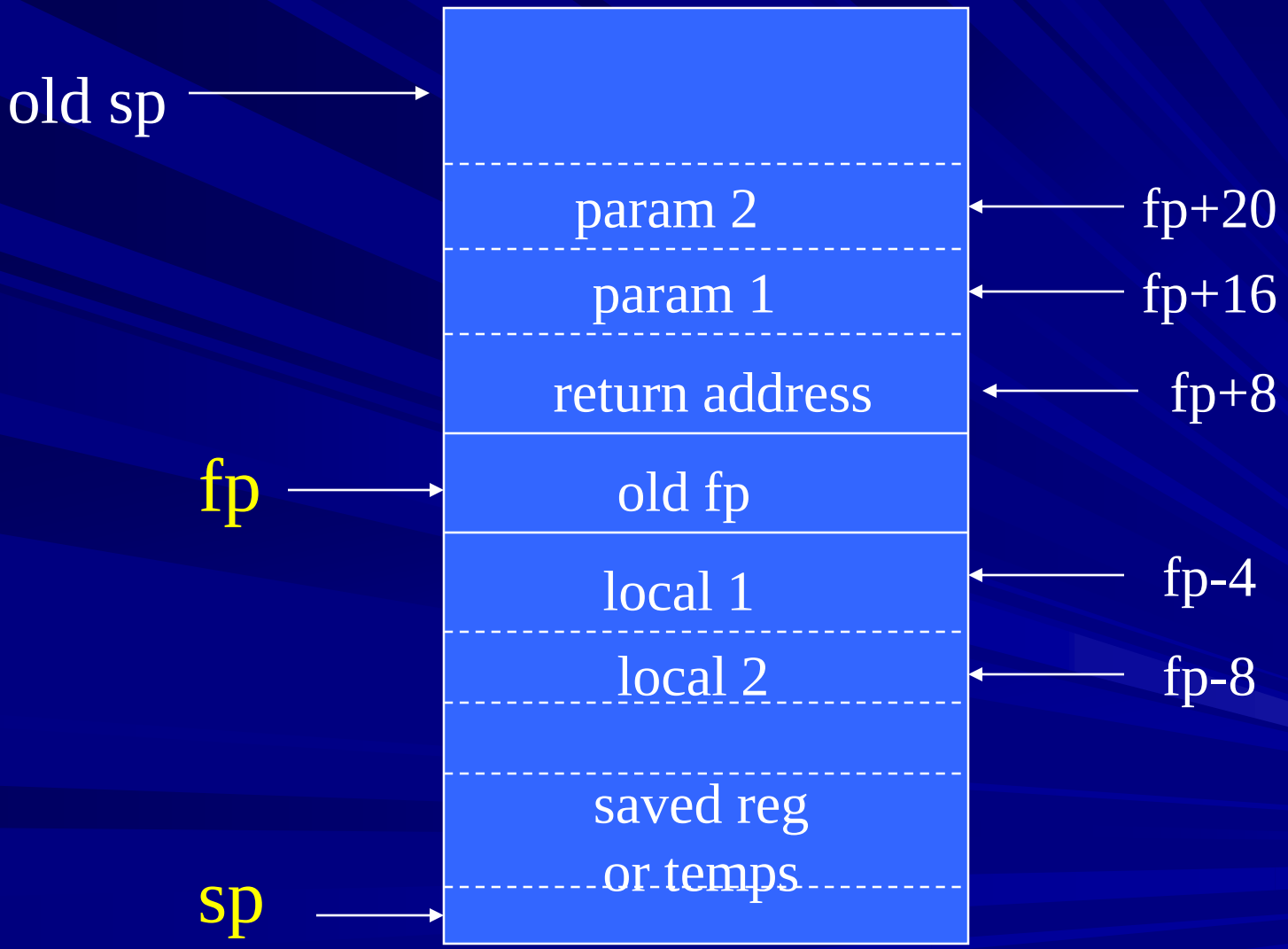
```
.data
_gid:    .word 0
.text
main:
        str x30, [sp, #0]
        str x29, [sp, #-8]
        add x29, sp, #-8
        add sp, sp, #-16
        ldr x30,  =_frameSize_MAIN
        ldr w30, [x30, #0]
        sub sp, sp, w30
        str x9, [sp, #8]
        str x10, [sp, #16]
        ……..
_begin_main:
        mov w9, #1
        ldr x10, =_g_gid
        str w9, [x10, #0]
_end_main:
        Epilogue
.data
  frameSize_MAIN: .word 8
```

1) When is the end of declaration? We are using procedure-level AR. Declarations in every block count.

2) Variable initializations in declarations will cause instructions to be generated.

3) Temporaries may be introduced during expression evaluations. AR size cannot be determined that easily.

Prologue Code Sequence



AARCH64 example

```
int gid;
int main() {
   gid = 1;
}
```

```
.text
main:
        Prologue
_begin_main:
        mov w9,  #1
        str w9, [x29, #-4]
_end_main:
        Epilogue
.data
_framesize_main: .word 4
```

Original:
str w9, [x10, #0]

Should we store w9 on AR?

No! Why?

w9 is called saved (clean)

ARMv8 Registers

Argument registers (X0-X7)

Caller-Saved temporary registers (X9-X15)

Callee-Saved registers (X19-X29)

X8, X16-X18, X29,X30 have special purpose

X8 – indirect result register (e.g. a function returns a structure)

X16-X17 – for linker

X18 – for platform ABI

X29 – fp

X30 – LR (Link register)

Data Structures

Add attributes to symtable.h

add “int offset” for *symbol table record*

add “int place” for *AST node* (for temporaries)

Global data: (*Avoid using global var if you want to support parallel execution*)

```
int ARoffset=4;
int reg_number = 5;
```

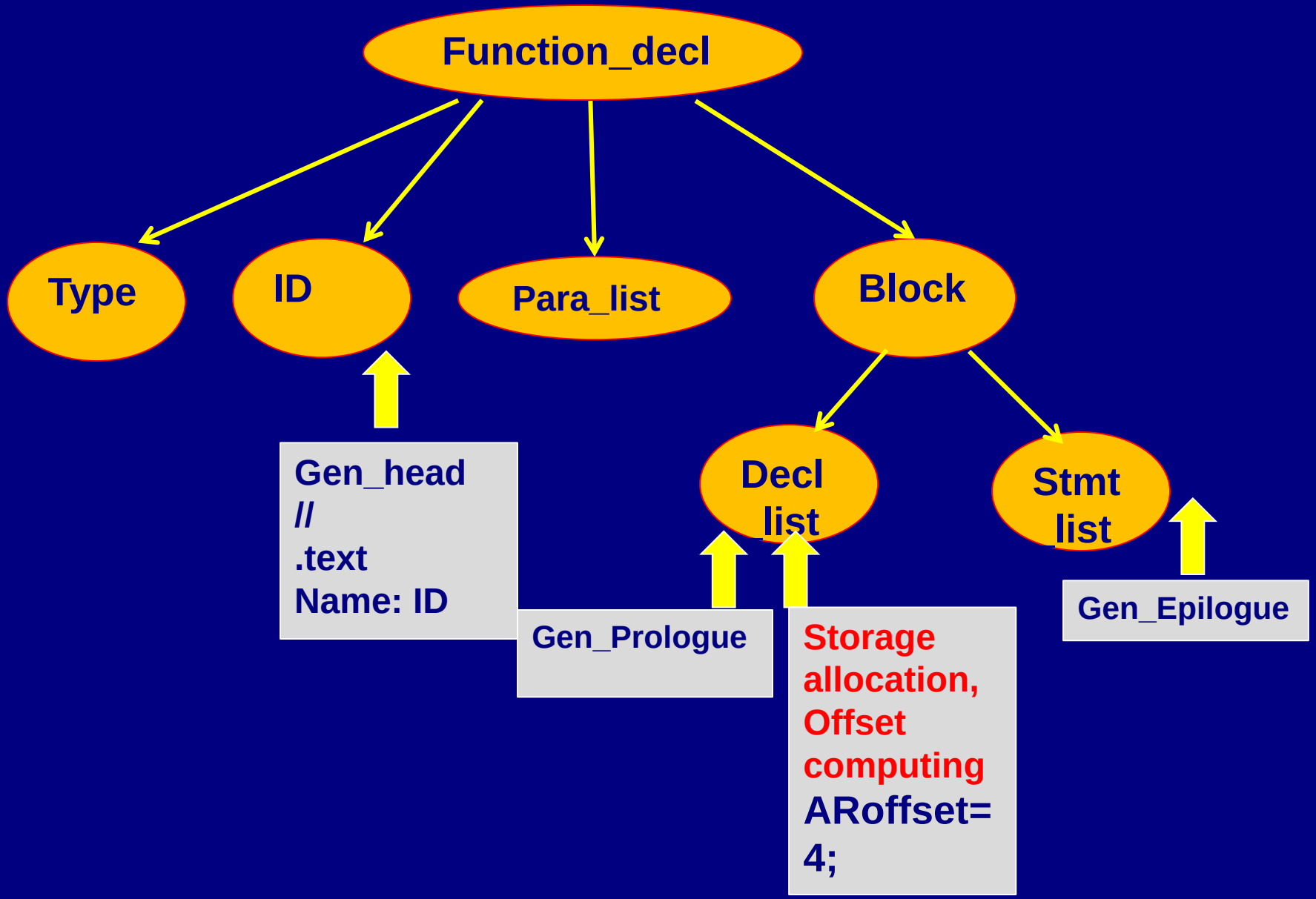
Routines:

gen_prologue(), gen_head(),
gen_epilogue(), get_reg(), get_offset

Local Variable Declaration

In routine symtable_add(), add

```
temp->offset = AReoffset;
AReoffset  += size; // e.g. size is 4
                    // if it is an array, the size of
                    // the array is used.
                    // if it is a struct, struct size is
                    // used
```


Function_decl
Type
ID
Para_list
Block
Gen_head
//
.text
Name: ID
Decl list
Stmt list
Gen_Prologue
Storage allocation, Offset computing
ARoffset= 4;
Gen_Epilogue

Genhead

```
gen_head (char *name):
{
   printf(“.text”);
   printf(“%s:\n”, name);
}
```

Prologue

```
gen_prologue (char *name):
    str x30, [sp, #0]              // store return address
    str x29, [sp, #-8]       // save old fp
    add x29, sp, #-8         // new fp
    add sp, sp, #-16         // new sp
    ldr x30, =_frameSize_name
    ldr w30, [x30, #0]
    sub sp, sp, w30                // push new AR
        .....
_begin_main:
```

Epilogue

gen_epilogue (char *name):

output the following instructions:

```
_end_name:
    ldr x30, [x29, #8]      // restore return address
    add sp, x29, #8         // pop AR
    ldr x29, [x29,#0]       // restore caller (old) fp
    RET x30
.data
_framesize_name:   .word   size
```

```
Procedure visit(SymReferencing n)
{
    int offset, reg;
    reg=get_reg();
    n.place = reg;
    offset = get_offset(n.symtbl_ref);
    if (local)
        printf("\t ldr    w%d, [x29, #%d]\n", reg, offset);
    if (global)  ……
}
```

```
Procedure visit(Constant n) {
    int offset, reg;
    reg = get_reg();
    n.place = reg;
    if (integer)
        printf("\t mov    w%d, #%d\n", reg, n.value);
    if (float) … /* save the FP constant in static
                memory with a label */
    if (string) … /* save the string constant in static
                  memory with a label */
}
```

```
Procedure visit(Computing n)
{
    int reg1, reg2, reg3;
    CodeGen(n.left_child);
    CodeGen(n.right_child);
    reg1 = n.right_child->place;
    reg2 = n.left_child->place;
    reg3 = get_reg();
    switch(n.op) {
    “ADD”: printf("\t add   w%d, w%d, w%d\n",
            reg3, reg1, reg2);
    “SUB”: …… }
}
```

```
Procedure visit(Assignment n)
/* stmt : id  = relop_expr; */ {
    int offset, reg;
    reg = n.right_child->place;
    offset = get_offset(n.left_child->symtbl_ref);
    if (local) printf("str w%d, [x29, #-%d]\n",
       reg, offset);
    if (global) … ?
}
```

```
ldr   x9, =global
str w8, [x9, #0]
```

```
get_reg() {
    return(reg_number++);
   /* how do you recycle registers? */
                }
```

get_offset()

gets the offset of the local variable from the symbol table.

```
get_reg() {
    return(reg_number++);
    /* how do you recycle registers? */
    /* if no CSEs (i.e. trees and no DAG), whenever
     a reg is used, it is freed. */
}
```

get_offset()

gets the offset of the local variable from the symbol table.

Making it more interesting

```
int main() {
   int lid;
   lid = 1;
   write(lid);
}
```

```
ldr w9, [x29, #-4]
mov w0, w9
bl _write_int
```

Code Generation Project

1. Arithmetic expressions
2. Assignments
3. Control structures (If, If-else, while)
4. Read/Write
5. Function calls (with no parameters)
6. Array references
7. Type conversions
8. Boolean expressions
9. Control structures (For loops)
10. Function call (with parameters)

Extra Credits

- Using caller/callee register convention
- Using register tracking to avoid redundant loads/stores
- Using value numbering to capture CSE's

Assignments and Expressions

x = y

if x,y are static variables

```
ldr    x9, =_y
ldr    w10, [x9, #0]
ldr    x9, =_x
str    w10, [x9, #0]
```

if x,y are locals (allocated on AR)

```
lw     w9, [x29, #offset-of-y]
sw     w9, [x29, #offset-of-x]
```

Assignments and Expressions

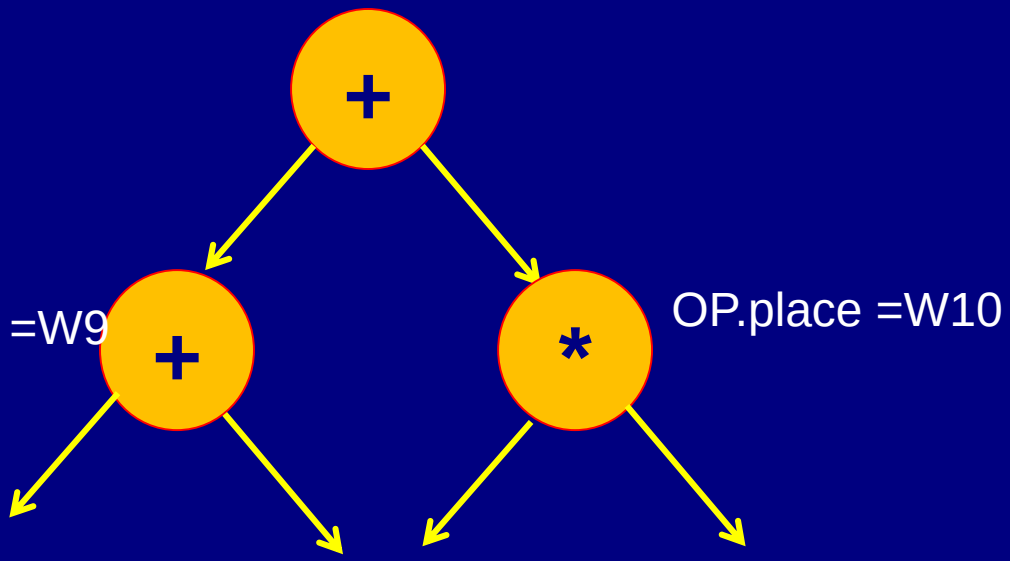
a = b + c (id ‘=‘ expr ‘;’)

assume a, b are locals and c is static

```
    ldr   w9, [x29, #offset-of-b]
    ldr   x10, =_c
    ldr   w11, [x10, #0]
    add   w9, w9, w11
    sw    w9, [x29, #offset-of-a]
```

Assignments and Expressions

OP.place = getreg();



Generate **Add W11, W9, W10**

To recycle registers, free(w9), free(w10)

Generate **Add W9, W9, W10**

Example of massive redundant loads/stores

A = B*B + C*C

D = A – B*C

7 ldr and 2 str generated

```
ldr w9, B
ldr w10, B
mul w9, w9, w10
ldr w10, C
ldr w11, C
mul w10, w10, w11
add w9, w9, w10
str w9, A
```

```
ldr w9, B
ldr w10, C
mul w9, w9, w10
ldr w10, A
sub w9 w10, w9
str w9, D
```

Only 2 ldr and 2 str needed

Assignments and Expressions

- Issues
  - Register assignment and replacement.
    - A typical IR code generator does not deal with registers.
    - One common approach is to assign pseudo registers first, then map unlimited pseudo registers to real registers.
    - One pass compiler needs to assign registers during code generation.
  - Loads/stores reduction.
    - Register Tracking

Simple code gen + optimizer

- If there is a separate optimization phase, the code generator can be much simplified.
  - *but the complexity moves to the optimizer.*
- The optimizer can
  - eliminate redundant loads/stores,
  - identify CSE (Common Sub-Expressions),
  - allocate local variables to registers, and
  - map unlimited number of pseudo registers to the limited real registers.

Register Tracking

- A simple local register allocation scheme which tracks the contents of allocatable registers.
- This simple allocation scheme can reduce redundant loads/stores, and efficiently recycle temporary registers.

Register Tracking (cont.)

- Register association list: for each operand register, we maintain the variable or temporary name it contains, if any.
- Each variable or temp name associated with a register has one status flag:
  - S (to be saved) or NS (not to be saved)

    (i.e. S means *dirty* and NS means *clean*)
  - Status S means a store is needed to save the value.

Register Tracking (cont.)

- live register: may be referenced again
- a temporary is dead once it is used.
  - A dead temporary shall be removed from the association list.
  - In other words, when we call free_reg(), we remove associations for temporaries.
- The cost of deallocating a variable or temporary from a given register is
  - 1: if its status is (NS)
  - 2: if its status is (S)
- A register has no associations costs 0 to allocate.

Example

A = B*B + C*C (assuming only 3 available regs)

	Register Association		
	W9	W10	W11
ldr w9, B	B (NS)		
mul w10,w9,w9		Temp(S)	
ldr w11, C			C(NS)
mul w9,w11,w11	Temp(S)		
add w11,w9,w10	freed	freed	A(S)

Example

B = B*B + C*C

	Register Association		
	W9	W10	W11
ldr w9, B	B (NS)		
mul w10,w9,w9		Temp(S)	
ldr w11, C			C(NS)
mul w11,w11,w11			Temp(S)
Add w9,w10,w11	B (S)	freed	freed

GetReg()

choosing the cheapest possible register

- e.g. a register with 0 allocation cost.
- If GetReg is used in a separate local optimizer, it can also choose the register that minimizes register spill cost.
- will be discussed in Ch. 13.

- uses the next use information to determine which register is cheaper to be freed (displaced).
- Here we use a simplified GetReg for an one-pass code generator.

Simple GetReg()

function GetReg()

if there exists a register R with no association,
choose R

else

choose the reg R with minimum allocation cost.

Save the value of R for any variable with status (S).

Register Tracking

- A simple local register allocation scheme which tracks the contents of allocatable registers.
- This simple allocation scheme can reduce redundant loads/stores, and efficiently recycle temporary registers.

Register Usage Convention

- Registers are partitioned into
  - caller saved (x9 to x15) and
  - callee saved (x19 to x29).
- Arguments can be passed through registers x0 to x7.
- In your code generator, you do not need to follow the AARCH64 convention because your generated code is self-contained.
- Therefore, you may group all registers together as a single register pool, using either caller save or callee save scheme.

Revised Code Gen
x=y

When y is an ID or a var_ref,

If y is not in a reg,

call R1=GetReg() and

generate a load of y and associate R1 with y(NS)

If x is not already in a register then

call R2=GetReg() and

generate a “move R2, R1” and

associate R2 with x(S).

Otherwise, y is in register R1 and x is in register R2,

so generate a “move R2, R1” and

associate R2 with x(S).

Examples

A = B*B + C

C = B

D = A + C

	Register Association			
	W9	W10	W11	W12
ldr w9, B	B(NS)			
mul w10,w9,w9		temp(S)		
ldr w11, C			C(NS)	
add w12,w10,w11		freed		A(S)
mov w11,w9			C(S)	
add w10,w11,w12	B(NS)	D(S)	C(S)	A(S)

Q: which register will be chosen by the next GetReg call?

Limitation of GetReg()

- in one-pass code generator

W9	W10	W11	w12
B(NS)	D(S)	C(S)	A(S)

The next GetReg call may choose w9, however,

- A,C may be dead (the current value is overwritten) and
- B may get referenced more times in the next few expressions,

so selecting w9 may be sub-optimal.

Limitation of GetReg()

- in one-pass code generator

W9	W10	W11	w12
B(NS)	D(S)	C(S)	A(S)

- In one-pass code generation, the compiler cannot see the *future references*.
- In a multi-pass code generator, GetReg can use future reference knowledge to perform *near optimal* local register replacement.

More Sophisticated GetReg

function GetReg()

- What to do if ***all registers are associated with temporaries*** ?

More Sophisticated GetReg

function GetReg()

- What to do if ***all registers are associated with temporaries*** ?
  - Must ***SPILL – store the victim temp into AR.***
  - What would happen to the .place attribute in semantic records?
- How do we know a register is spilled?

More Sophisticated GetReg

- What would happen to the .place attribute in semantic records?
- How do we know a register is spilled?

The attribute .place is now more than just a simple register number, but an association -- a (reg, name) pair.

When .place is referenced, must check if the name is still associated with the reg.

For example, if a temporary is spilled, a load must be issued to get the data back.

A Complex Example

A = b + (c*d+e*f*(g+h*j*(j+k…))))

If b is loaded to register w9, and later spilled (disassociated), what register to use when b+expr is reduced?

reload from memory location b??

This strategy does not always work, check the following example:

A = b + (….+ function1(…))

What if function1() modify variable b?

In this case, reloading from b for b+expr may be incorrect for some PLs!

A Cheaper Solution

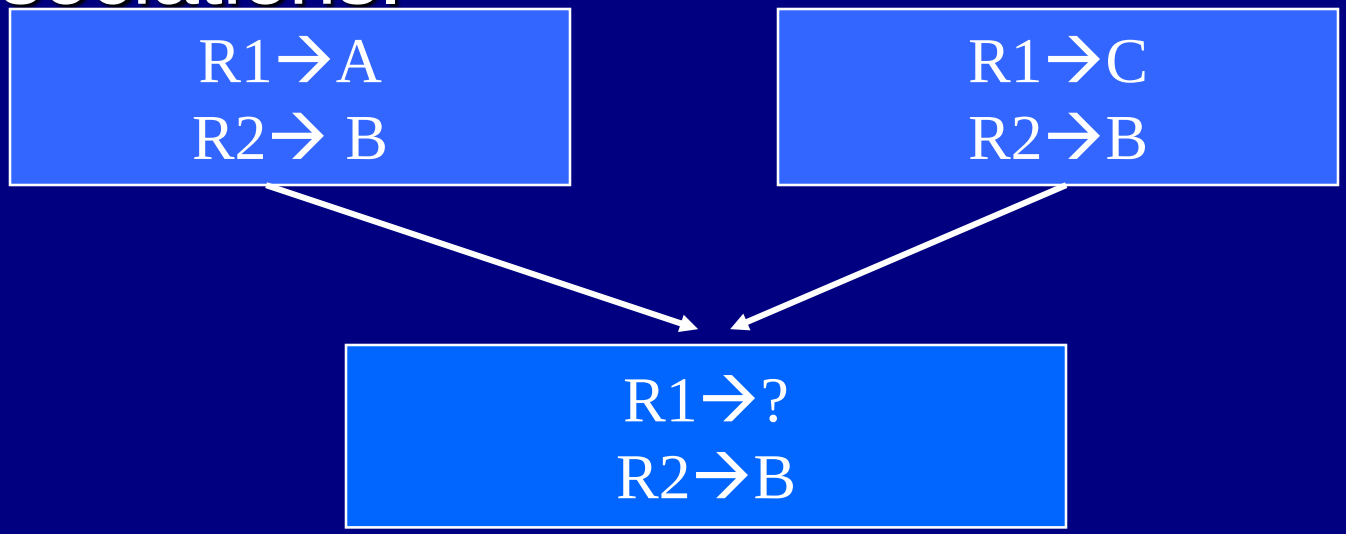
- Don’t associate global variables to registers.
  - In our compiler, local variables cannot be modified via side-effect of function calls.

Effect of Aliasing and Subroutine Calls

- There is no possible aliasing in our project.
- If aliasing is allowed
  - Simple approach: every possible pointer reference kills the current register association.
  - Tracking aliasing: compute a set of data objects that the pointer may alias with.
    - A pointer reference kills only the register association for the set of objects it may alias with.
- We may assume a subroutine call kills the register association.
  - So all registers with S status will be saved before the call.

Local Register Allocation

- We do not carry the register association list cross blocks.
- If a block has more than one predecessors, it is difficult to maintain consistent associations.



Caveats of Register Tracking

- No global allocation
- No aliasing
- No subroutine calls

Global Register Allocation

- One unique pseudo register for each local variable or CSE (common-subexpression)
- If not enough: only selected ones get reg
  - Usage count
  - Loop nesting level
- Modern approaches: (may be discussed in details later)
  - Graph coloring
  - Priority based graph coloring