Compiler Technology of Programming Languages

Code Generation

Arrays, Functions, Switch/Cases

Prof. Farn Wang

Array References

int a[100], x[200];

Assume

starting address of a[] is B.

width of array element is 4 bytes

The address of a[j] should be B + j*4

```
ldr  x4,    =a      # load base address of a[], j in r6
mul x5,x6,#4        # j*4,  lsl x5,x6,#2 is faster
add x5,x4,x5        # B+j*4
ldr  w5,[x5, #0]    # load a[j]
```

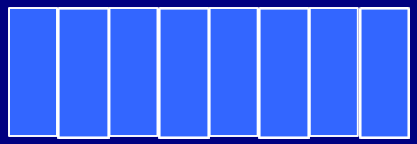
Array References (cont.)

int a[100][200];
 Assume
 starting address of a[] is B.
the address of a[j][k] should be
 B + j*200*4 + k*4 = B + (j*200+k)*4

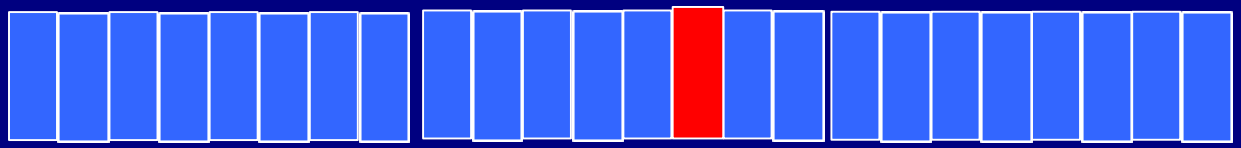
```
ldr   x4,      =a      # load base address of a[]
mul x5,x6,#200   # j*200, assume j in x6
add  x5,x7,x5    # j*200+k, assume k in x7
mul   x5,x5,4    # (j*200+k)*4
add  x5,x4,x5    # base B + (j*200+k)*4
ldr   w5,[x5, #0]   # load a[j][k]
```

Array Layout in C

A[8]



A[3][8]



A[1][5]

Array References (cont.)

int a[n1][n2][n3]…;

Assuming the starting address of a[] is ***B***.

the address of a[i1][i2][i3]…[ik] would be

B + i1*n2*n3*… +i2*n3*n4*… +…+ ik*4

➔ ***B*** + ((…(i1*n2+i2)*n3 + i3)*n4 …)* width

General form:

B + (***variable_part*** * width)

In short,

B + (Vp * w)

Array References (cont.)

int a[n1][n2][n3]…;

B + ((…(i1*n2+i2)*n3 + i3)*n4 …)* width

Verify this formula:

int a[n1];

a[i] ➔ ***B*** + (i* width)

int a[n1][n2];

a[i][j]➔ ***B*** + ((i*n2)+j)* width)

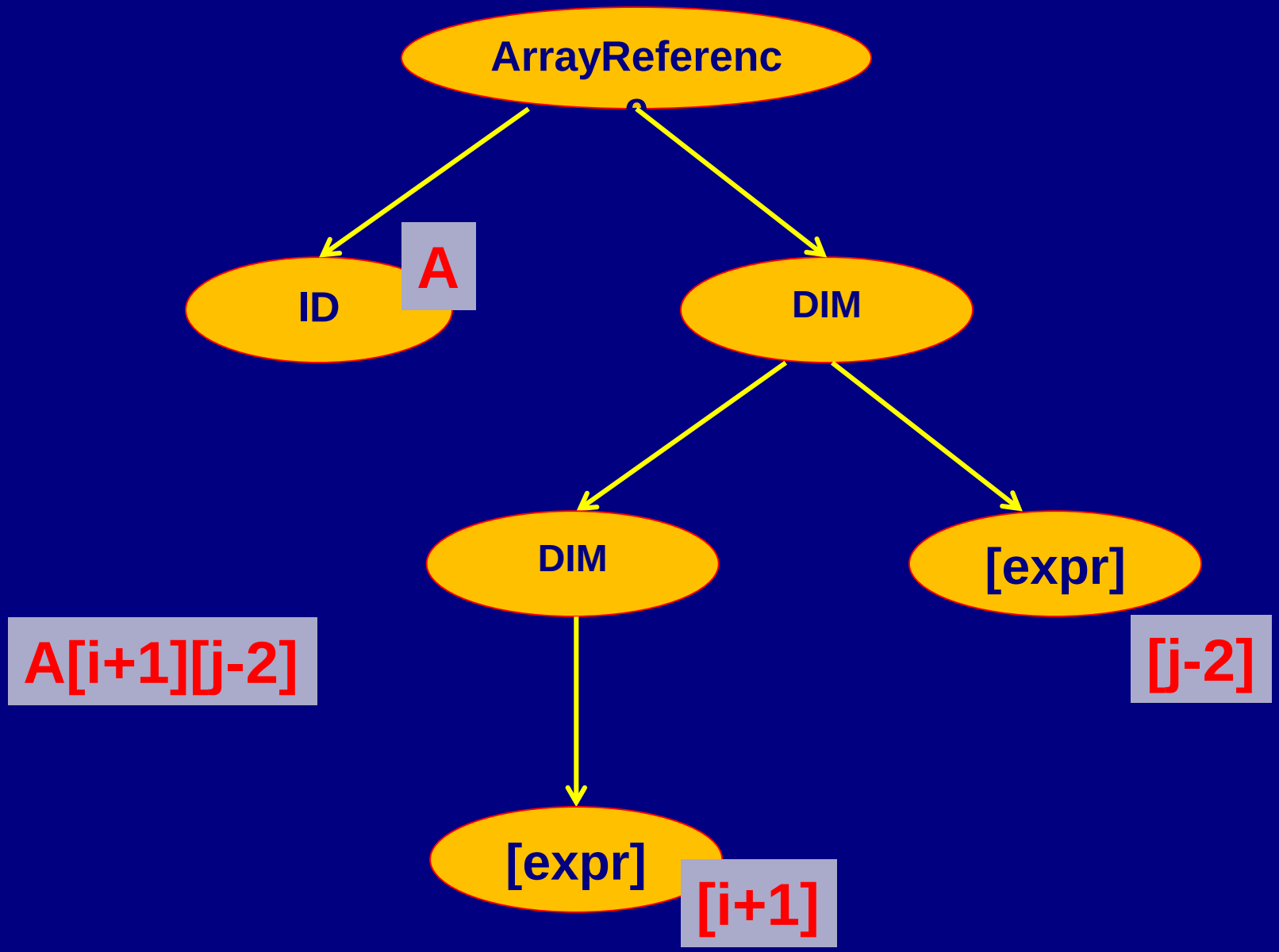
int a[n1][n2][n3];

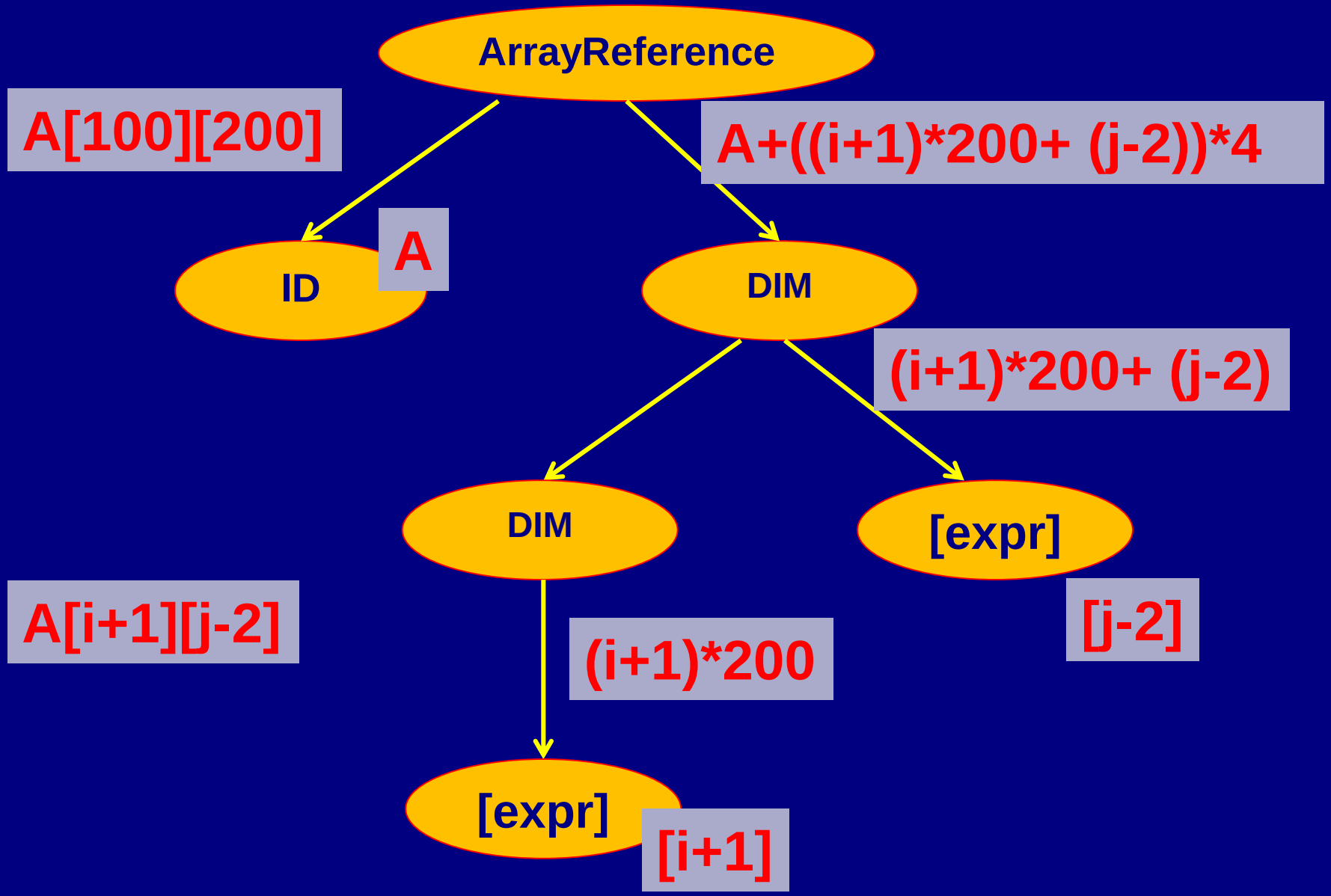
a[i][j][k]➔ ***B*** + (((i*n2)+j)*n3)+k)* width)

Important information:

Base address: ***B***, Dimensional size: n1, n2, n3, …

Element size: width


ArrayReferenc
ID
A
DIM
DIM
[expr]
[expr]
[i+1]
A[i+1][j-2]
[j-2]

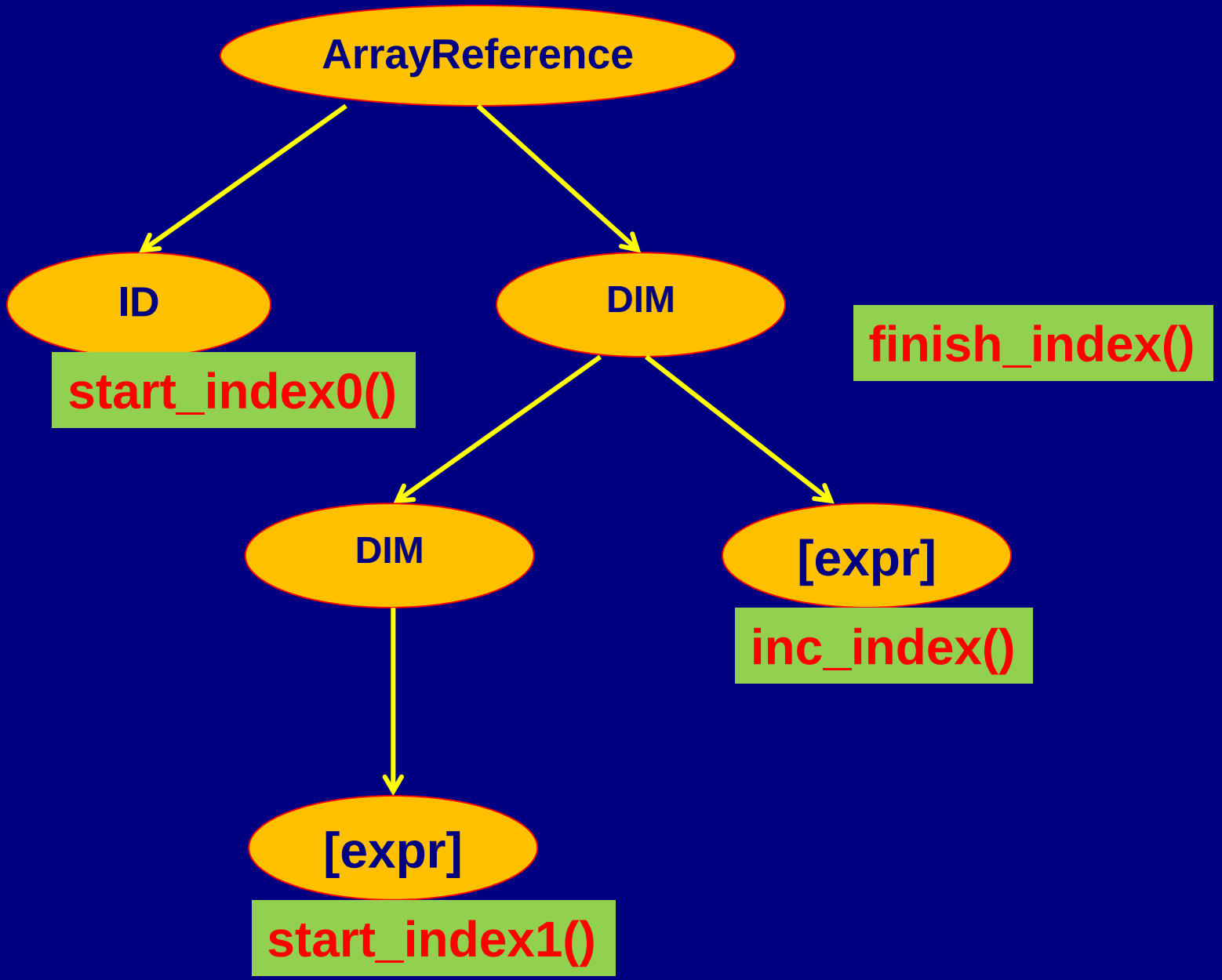

ArrayReference
A[100][200]
A+((i+1)*200+ (j-2))*4
ID
A
DIM
(i+1)*200+ (j-2)
DIM
[expr]
A[i+1][j-2]
(i+1)*200
[j-2]
[expr]
[i+1]

Adding Action Routines

- factor : ID dim {finish_index();}
- stmt : ID dim {finish_index();} = relop_expr;

dim : [expr] {start_index();}
| dim [expr] {inc_index();}

start_index: prepare for array information
inc_index: computing the Varible_Part
finish_index: computing the address of this array ref.


ArrayReference
ID
start_index0()
DIM
finish_index()
DIM
[expr]
inc_index()
[expr]
start_index1()

Semantic record for non-terminal dim

```
struct IndexType {
    int cnt;    // how many subscripts have been processed
    int  Vp;  // address of variable_part of indexed array
    int base_address; // offset? how to represent global?
                                  //  this part could be optional.
    int dim_size[C];  // C is max dimensions to support
    int dims; // declared info }
```

start_index() will check symbol table entry for ID and *obtain the information of dimension and dim sizes to fill the semantic record*.

Array References in C– (cont.)

start_index0:

get information of ID

AST.idx=malloc(IndexType);

AST.cnt = 1;

AST.base_address = ID.base_address,

get dim_size[] and dims info;

start_index1:

AST.VP = expr.place;

Why not load base address now? *May tie up a register too long*

Array References in C– (cont.)

inc_index:

AST.cnt++; check if cnt is valid

generate code for

AST.VP=

AST.dim_size[AST.cnt-1] * AST.Vp + expr.place;

finish_index:

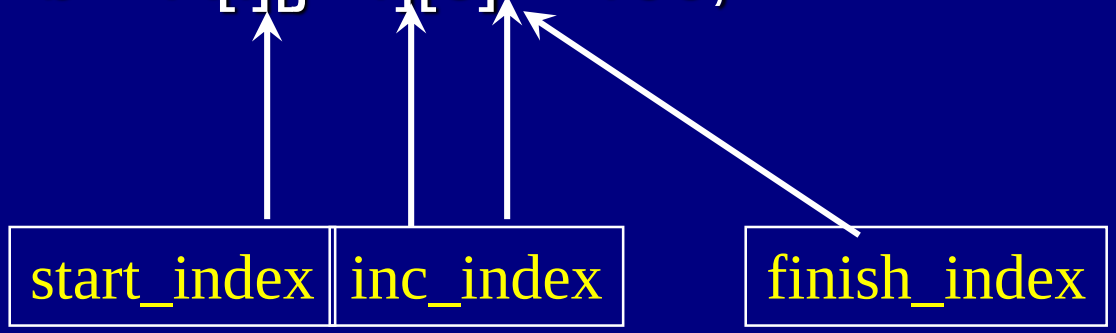
generate code for (Vp = Vp*w + base_address)

Example

int A[10][20][30];

how to process reference A[i][j+1][5]

b = A[i][j+1][5] + 100;



start_index:

get information about A

get a reg, x6, move i into x6

$$->cnt = 1, $$->base_address = A or offset_a(fp)

$$->Vp = x6

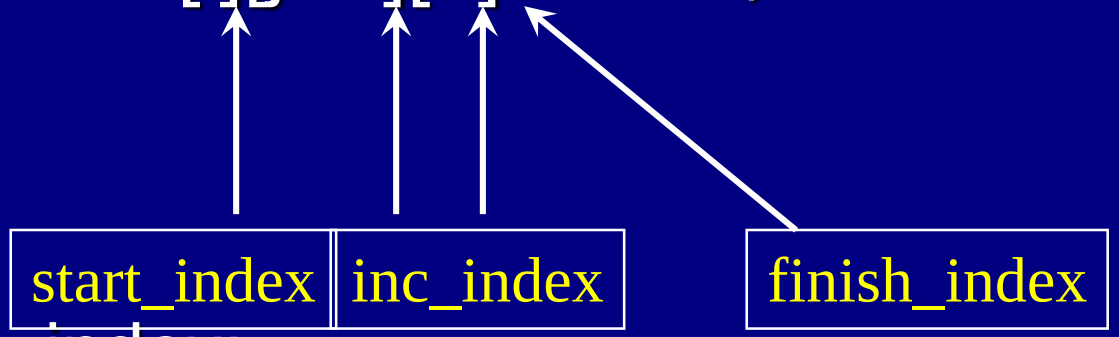
dim	3
dim_size	10,20,30

Example

int A[10][20][30];

how to process reference A[i][j+1][5]

b = A[i][j+1][5] + 100;



Inc_index:

cnt=2,

gen code “mul r6, r6, #20”

mov r11, #20
mul r6, r6, r11

gen code “add r6, r6, r5” // assuming r5 has (j+1)

dim	3
dim_size	10,20,30

Inc_index:

cnt=3,

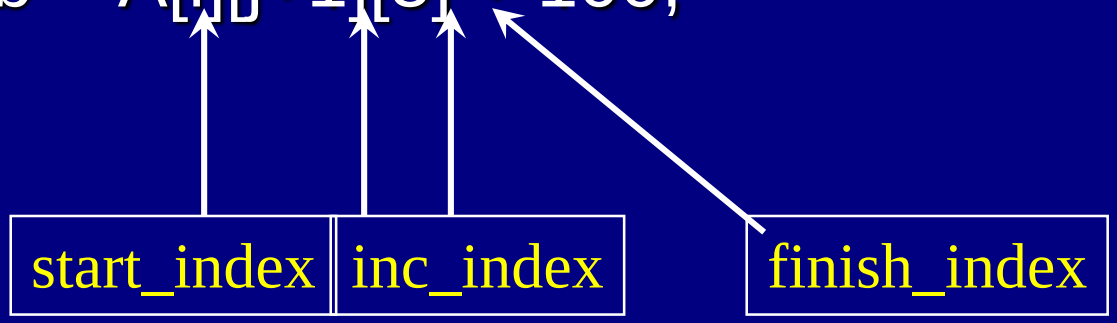
gen code “mul r6, r6, #30” “add r6, r6, #5”

Example

int A[10][20][30];

how to process reference A[i][j+1][5]

b = A[i][j+1][5] + 100;



finish_index:

gen code “mul x9, x9, #4”

VP*width

get a temp reg x10, gen code “ldr x10, =A

Get base addr

gen code “add x10,x9,x10” // add Vp (variable part) to base

free_reg(x9), get a reg, say, x11, $1->place=x11

gen code “ldr x11, [x10, #0]”

Translating Procedure/Function

- Procedure/function declaration
  - Construct a symbol table entry
  - Attribute record (ret type, param list, …etc)
  - Prologue/Epilogue code for the procedure body
- Procedure/function call
  - Using attribute information set up from the declaration
  - Parameter checking and passing
  - Transfer of control (bl)

Processing Calls with Parameter Lists

- ID ‘(‘ relop_expr_list ‘)’ {

```
check_param_type($1, $3);
save_registers();          // optional
passing_param();
gen_proc_call(name);
gen_after_return();        }
```

Check_param_type:

1. get the param_list of procedure ID from the symbol table
2. check if the number of arguments and argument types match with the procedure’s parameter list. Insert type conversion instructions if needed.

Calls with Parameter Lists (cont.)

save_registers:

if register convention is used

{save any caller-save registers whose values are to be preserved cross the call; save parameters in registers}

passing_param:

push space on the stack for arguments

if register convention is used

{ copy up to 4 parameters into $r0 to $r3;

copy the remaining arguments to the stack;}

else

copy all the arguments to the runtime stack;

Example

```
proc_a(x,y,z)
        .....
        proc_b(b+c, a*3);
        .....
```

Instructions to evaluate the two parameters: b+c and a*3, must have been generated, and the results can be put in $r0, and $r1. Two words should be allocated on stack for the two parameters.

Processing Calls with Parameter Lists (cont.)

gen_proc_call(name):

- generate “bl name” instruction, in our project, name is the label of the procedure.

gen_after_return():

- generate a copy instruction that moves the return value from $r0 to a target register.
- remove stack space allocated for arguments
- restore saved registers if any

If the returning object is larger than what can be stored in r0-r3, a pointer is returned

Processing Calls with Parameter Lists (cont.)

Example: Foo (I+J, K)

```
ldr   r4, I
ldr   r5, J
add r6,r4,r5
ldr   r4,K
add     sp,sp,#-8          // push space for args
mov r0, r6
mov r1, r4
bl        _Foo
mov r5,r0                  // returned value
add    sp, sp, #8          // discard arguments
```

Evaluation

Passing

Call

What if we do not pass arguments in registers?

Remember we should push arguments in reverse order !!

What if a caller-save register is used to store a*b? say r12

Example: **a*b** + Foo (I+J, K)

```
ldr   r4, I
ldr   r5, J
add r6,r4,r5
ldr   r4,K
```

Evaluation

str r12, [fp, #- tmp_offset]

```
add    sp,sp,#-8        // push space for args
mov r0, r6
mov r1, r4
```

Passing

```
bl       _Foo
```

Call

```
mov r5,r0          // returned value
add   sp, sp, #8          // discard arguments
```

ldr r12, [fp,#- tmp_offset]

Translating Return

return RETexpr ;

{copy ret value to $r0;

gen_return();}

gen_return()

generate “j _end_of_foo” where foo is the name of the procedure/function

Leaf routine optimization

- The simplest leaf routines (e.g. standard math functions) don't use the stack at all:
  - Take arguments in registers
  - Compute entirely in caller save registers
  - Return results in registers
- A set of important library routines can be implemented this way.
- ARM could use x0-x7 as caller save reg for leaf routines.

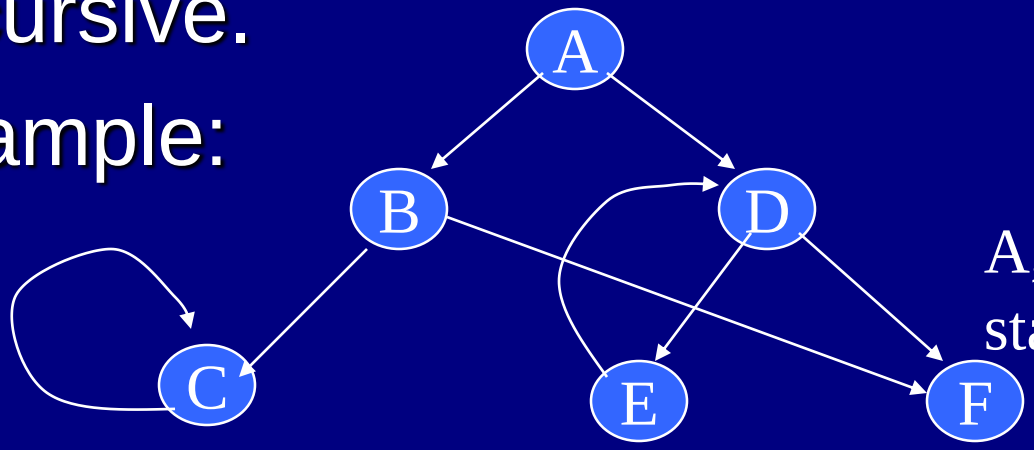
Minimizing Calling Overhead

- Procedure In-lining
- Using calling convention
- Leaf routine optimization
- Non-recursive routine optimization

Non-recursive routine optimization

- Non-recursive routines can have their AR statically allocated.
- We can build a call graph, any subprogram that appears on a cyclic path in a call graph is potentially recursive.

  example:



A,B,F AR can be statically allocated

Case/Switch statement

```
swicth ( c )
{
    case 0: stmtA;
    case 2: stmtB;
    case 5: stmtC;
    ….
    default: stmtD;
}
```

Switch is used frequently in interpreters/simulators.

The main motivation for the Switch statement is for efficient target code, not for syntactic elegance

Generating efficient code for switch stmt is challenging.

Case statement in Modula-2

```
CASE expression of
        0 : stmtA;
        2 : stmtB;
        3..5 : stmtC;
        7,10: stmtD;
        ELSE: stmtE;
END
```

Allows value ranges

Code Gen for Switch statement

- Cascaded if

  if (c == 0) goto stmtA

  else if (c==2) goto stmtB

  else if (c==5) goto stmtC

  ….

- Search table

  the compiler generate code to look up the case value in the table, if a value is found, the matching goto statement is executed. Linear, binary, and hashing search methods can be used.

0	goto A
2	goto B
5	goto C

Code Gen for Switch statement (cont.)

- Jump table

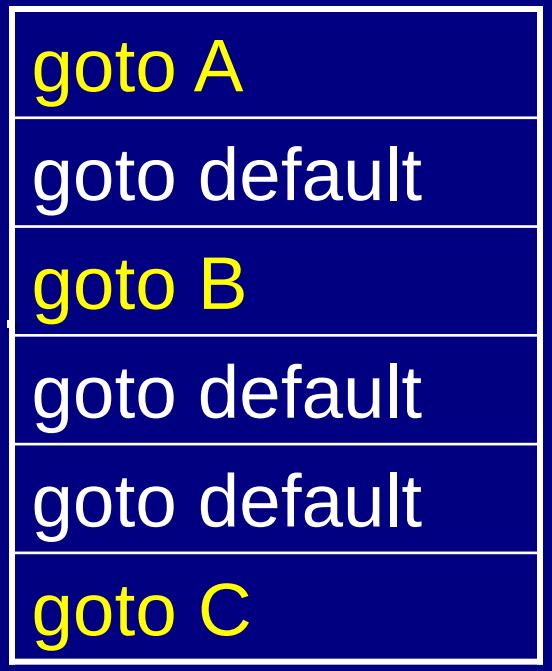
  A jump table is an array of jump instructions, indexed by case value.

  The compiler processes the switch with a single test and an array access.

  if (argument c is in range)

  goto jump_table[c – smallest case value];

  else goto default;



Code Gen for Switch statement (cont.)

- Jump table

  A jump table can also store the address of the target block. In this case, an indirect jump is used.

Label A
Label default
Label B
Label default
Label default
Label C

if (argument c is in range)

goto *jump_table[c – smallest case value];

else goto default;

Code Gen for Switch (cont.)

- Cascaded if is not as dumb as it looks.
  - For highly biased cases.
  - For a small number of cases (say, 2-4)
- Jump table is generally efficient, but may waste much space when the case value range is not densely covered.
- The jump table approach uses indirect jump which may cause branch mis-prediction.
- A good compiler often uses all three alternatives: search table, jump table and cascaded ifs, and their combinations.

Switch Code Gen

- Large number of cases, dense value range → Jump Table
- Medium to large number of cases, sparse value range → Search Table (linear/binary/hashing)
- Small number of cases, and/or highly biased → Cascaded if's
- Large, dense, very biased → mix if's and jump table

Can you give a case that requires all three?

Some issues to consider

- One pass code gen:
  - Jump table can only be produced after all cases are handled.
  - In one-pass compiler, a jump that skips all cases to the selector code is generated.
  - The selection code picks the target label using the switch value, and jumps back to the selected case.
  - Usually only one code generation strategy is used since the number of arms and value range is not known until the complete Switch statement is processed.
- Need to handle nested switches.

Related Questions

- Pascal and standard C do not allow ranges in their label lists?
  - GNU C allows them.
- Standard Pascal does not allow “default” or “else” label for the default clause.
- If an expression is evaluated to a non-existing label, is this a static or dynamic error?

For Pascal: a dynamic error
For C and Fortran90: not an error,
the entire construct has no effect.