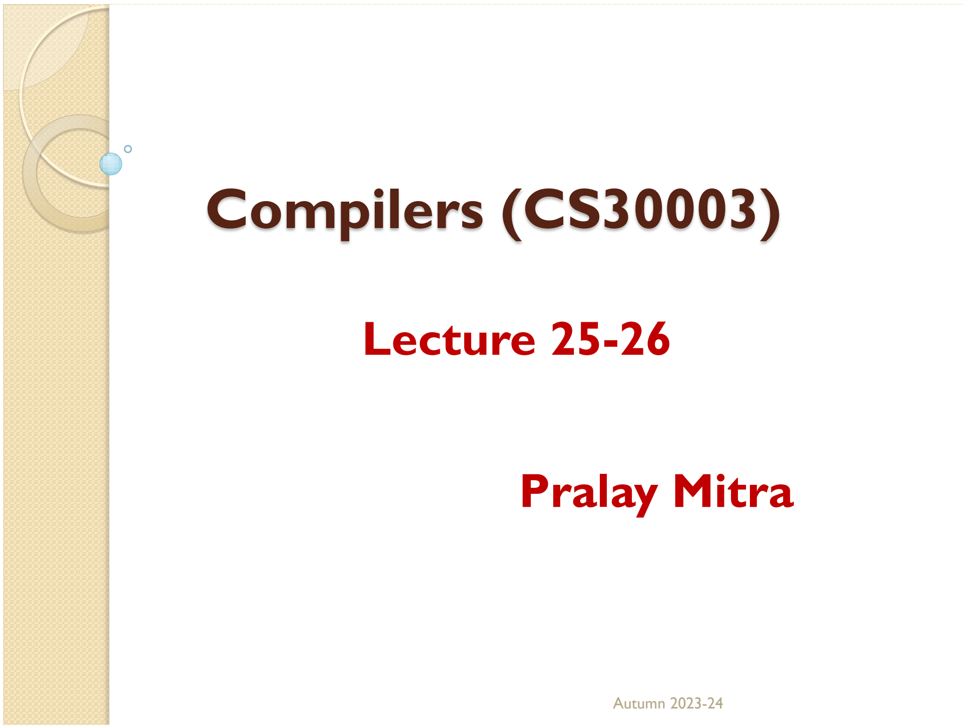
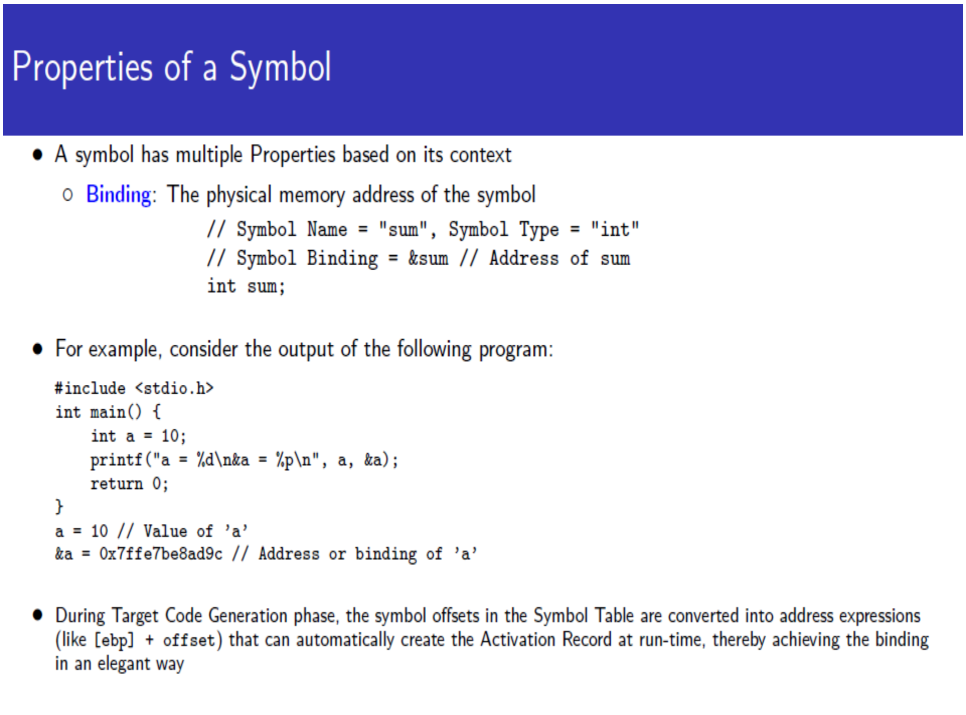


Compilers (CS30003)

Lecture 25-26

Pralay Mitra

Autumn 2023-24



Properties of a Symbol

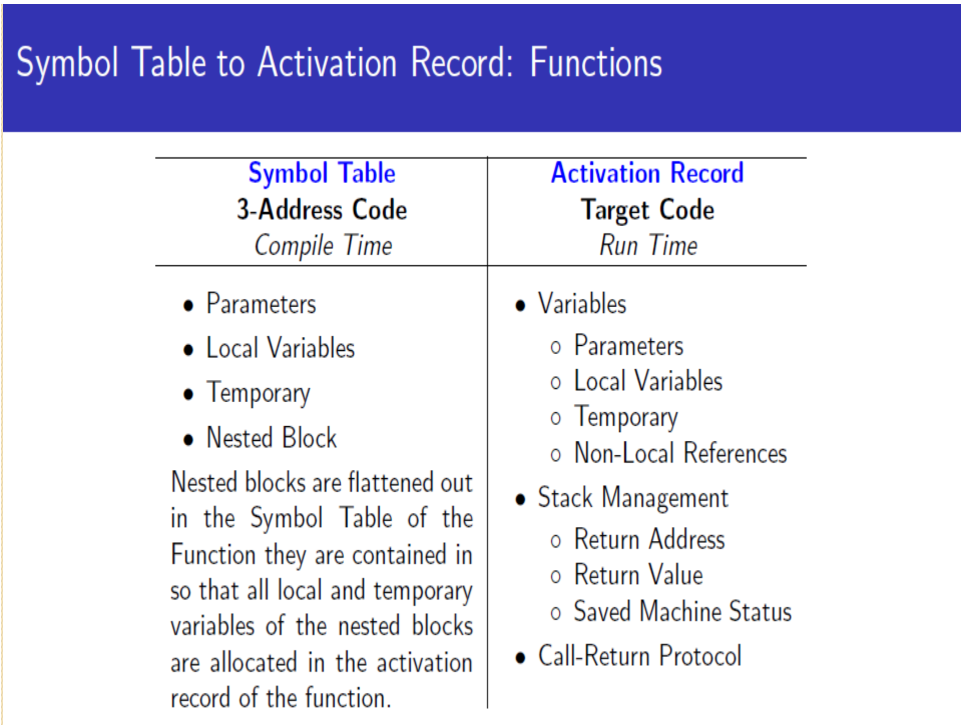
- A symbol has multiple Properties based on its context
  - Binding: The physical memory address of the symbol

```
// Symbol Name = "sum", Symbol Type = "int"
// Symbol Binding = &sum // Address of sum
int sum;
```

- For example, consider the output of the following program:

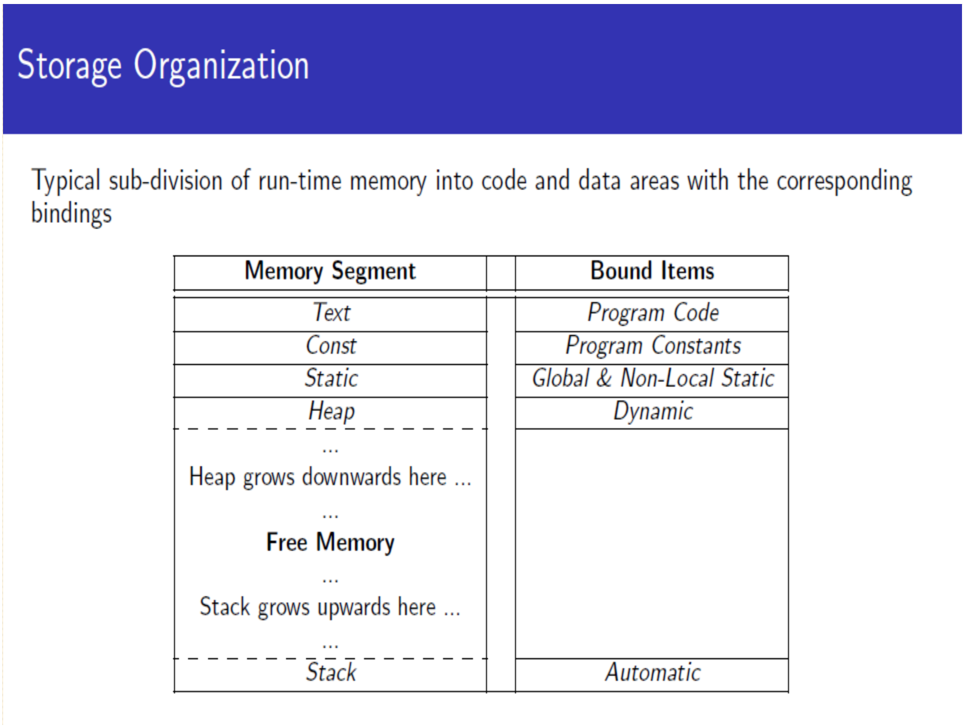
```
#include <stdio.h>
int main() {
    int a = 10;
    printf("a = %d\n&a = %p\n", a, &a);
    return 0;
}
a = 10 // Value of ’a’
&a = 0x7ffe7be8ad9c // Address or binding of ’a’
```

- During Target Code Generation phase, the symbol offsets in the Symbol Table are converted into address expressions (like `[ebp] + offset`) that can automatically create the Activation Record at run-time, thereby achieving the binding in an elegant way



Symbol Table to Activation Record: Functions

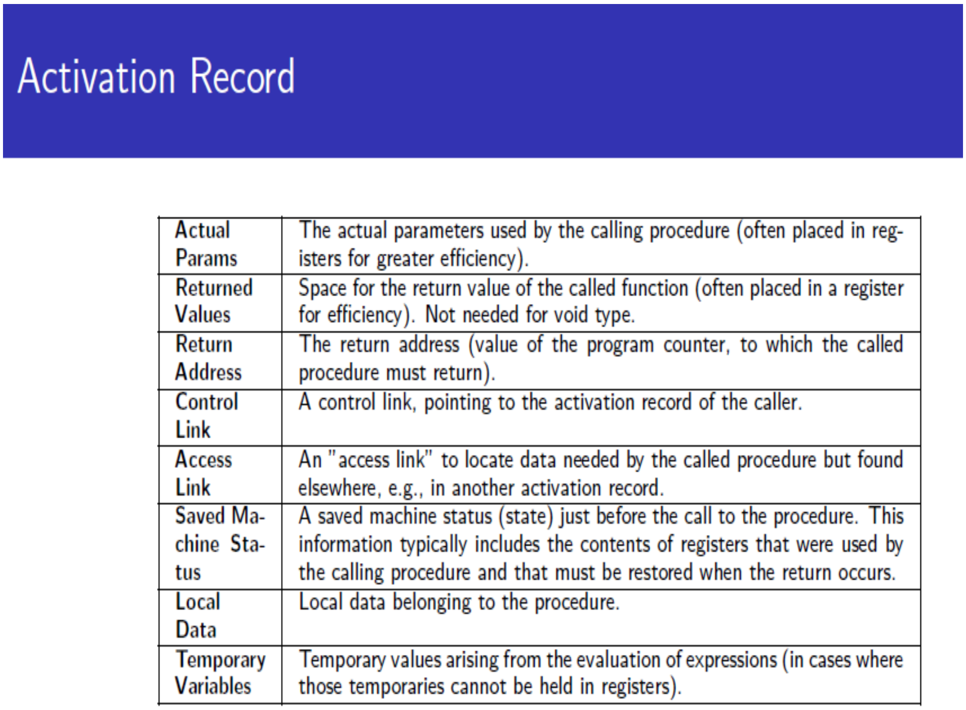
Symbol Table **3-Address Code** *Compile Time*	Activation Record **Target Code** *Run Time*
• Parameters • Local Variables • Temporary • Nested Block Nested blocks are flattened out in the Symbol Table of the Function they are contained in so that all local and temporary variables of the nested blocks are allocated in the activation record of the function.	• Variables ○ Parameters ○ Local Variables ○ Temporary ○ Non-Local References • Stack Management ○ Return Address ○ Return Value ○ Saved Machine Status • Call-Return Protocol



Storage Organization

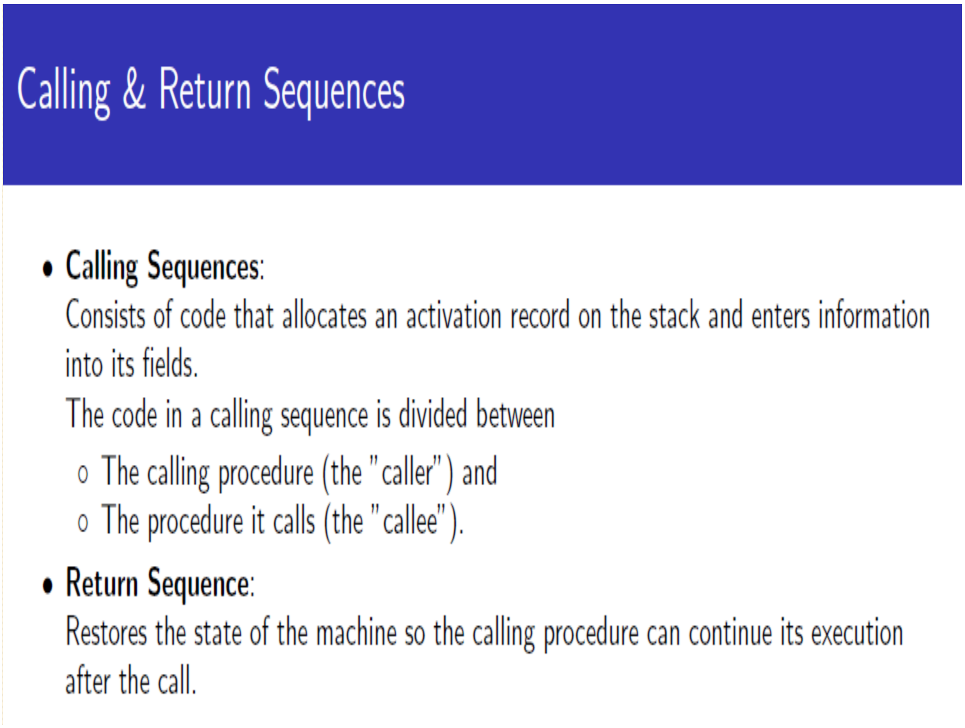
Typical sub-division of run-time memory into code and data areas with the corresponding bindings

Memory Segment		**Bound Items**
Text		*Program Code*
Const		*Program Constants*
Static		*Global & Non-Local Static*
Heap		*Dynamic*
... Heap grows downwards here ... ... **Free Memory** ... Stack grows upwards here		
Stack		*Automatic*



Activation Record

Actual Params	The actual parameters used by the calling procedure (often placed in registers for greater efficiency).
Returned Values	Space for the return value of the called function (often placed in a register for efficiency). Not needed for void type.
Return Address	The return address (value of the program counter, to which the called procedure must return).
Control Link	A control link, pointing to the activation record of the caller.
Access Link	An "access link" to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.
Saved Machine Status	A saved machine status (state) just before the call to the procedure. This information typically includes the contents of registers that were used by the calling procedure and that must be restored when the return occurs.
Local Data	Local data belonging to the procedure.
Temporary Variables	Temporary values arising from the evaluation of expressions (in cases where those temporaries cannot be held in registers).



Calling & Return Sequences

- **Calling Sequences**:
  Consists of code that allocates an activation record on the stack and enters information into its fields.
  The code in a calling sequence is divided between
  - The calling procedure (the "caller") and
  - The procedure it calls (the "callee").
- **Return Sequence**:
  Restores the state of the machine so the calling procedure can continue its execution after the call.

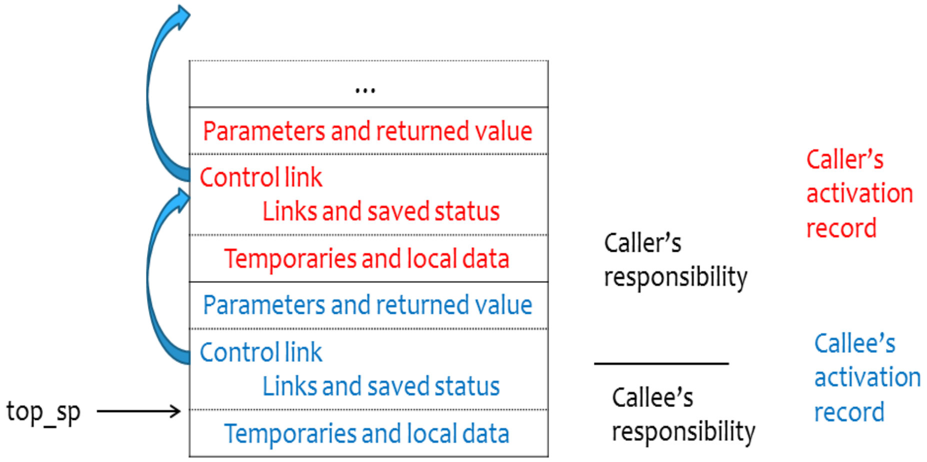
Calling Sequence

1. The caller evaluates the actual parameters.
2. The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to move past the caller's local data, temporaries and the callee's parameters and status fields.
3. The callee saves the register values and the other status information.
4. The callee initializes its local data and begins execution.

Return Sequence

1. The callee places the return value next to the parameters.
2. Using machine status field callee restores *top_sp*, registers and then branches to the return address that the caller placed in the status field.
3. Caller uses the return value as it knows where it is relative to the *top_sp*.

Calling Sequence



…

Parameters and returned value

Control link

Links and saved status

Temporaries and local data

Parameters and returned value

Control link

Links and saved status

Temporaries and local data

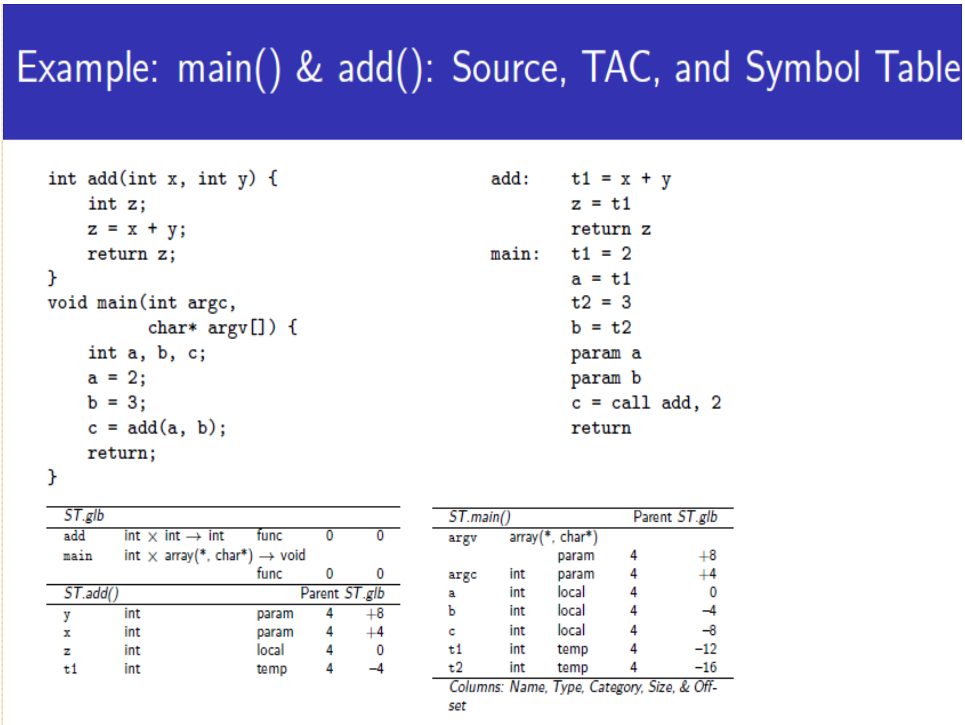
top_sp

Caller's responsibility

Callee's responsibility

Caller's activation record

Callee's activation record



Example: main() & add(): Source, TAC, and Symbol Table

```
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}
void main(int argc,
          char* argv[]) {
    int a, b, c;
    a = 2;
    b = 3;
    c = add(a, b);
    return;
}
```

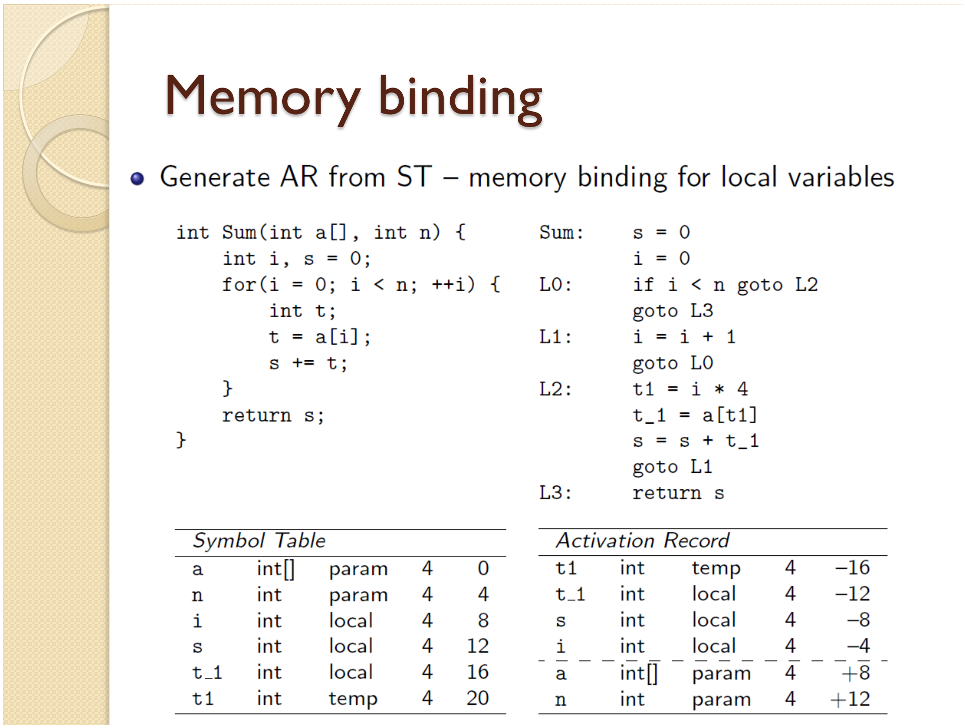
```
add:    t1 = x + y
        z = t1
        return z
main:   t1 = 2
        a = t1
        t2 = 3
        b = t2
        param a
        param b
        c = call add, 2
        return
```

ST.glb				
add	int × int → int	func	0	0
main	int × array(*, char*) → void	func	0	0

ST.add()		Parent *ST.glb*		
y	int	param	4	+8
x	int	param	4	+4
z	int	local	4	0
t1	int	temp	4	−4

ST.main()		Parent *ST.glb*		
argv	array(*, char*)	param	4	+8
argc	int	param	4	+4
a	int	local	4	0
b	int	local	4	−4
c	int	local	4	−8
t1	int	temp	4	−12
t2	int	temp	4	−16

Columns: Name, Type, Category, Size, & Offset



Memory binding

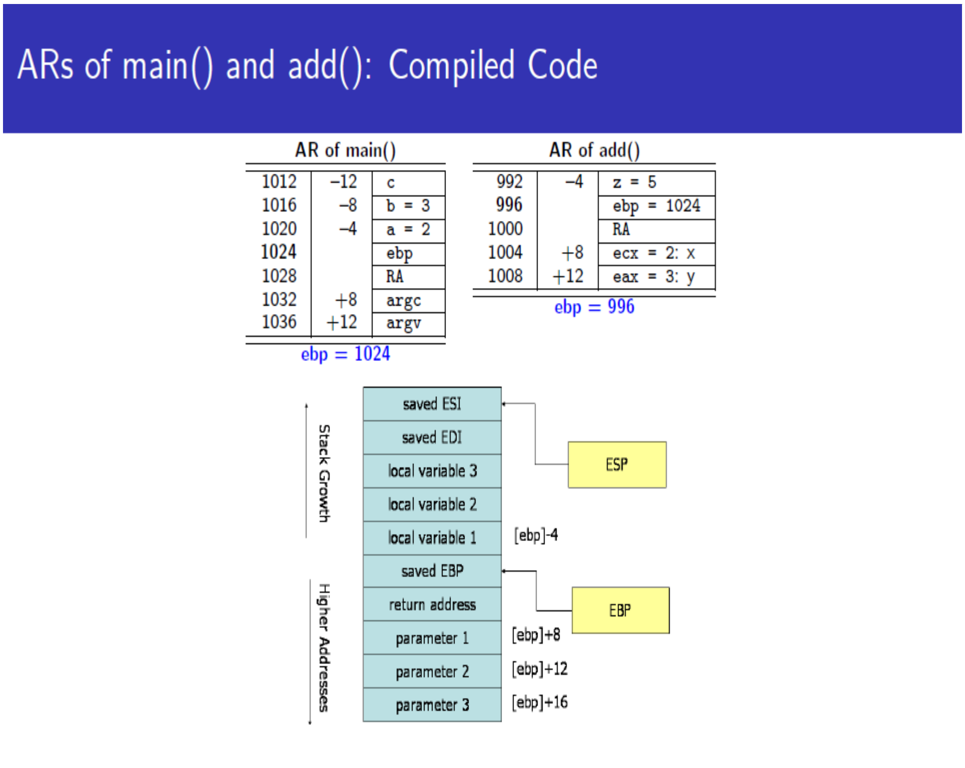
- Generate AR from ST – memory binding for local variables

```
int Sum(int a[], int n) {
    int i, s = 0;
    for(i = 0; i < n; ++i) {
        int t;
        t = a[i];
        s += t;
    }
    return s;
}
```

```
Sum:    s = 0
        i = 0
L0:     if i < n goto L2
        goto L3
L1:     i = i + 1
        goto L0
L2:     t1 = i * 4
        t_1 = a[t1]
        s = s + t_1
        goto L1
L3:     return s
```

Symbol Table				
a	int[]	param	4	0
n	int	param	4	4
i	int	local	4	8
s	int	local	4	12
t_1	int	local	4	16
t1	int	temp	4	20

Activation Record				
t1	int	temp	4	−16
t_1	int	local	4	−12
s	int	local	4	−8
i	int	local	4	−4
a	int[]	param	4	+8
n	int	param	4	+12



ARs of main() and add(): Compiled Code

AR of main()

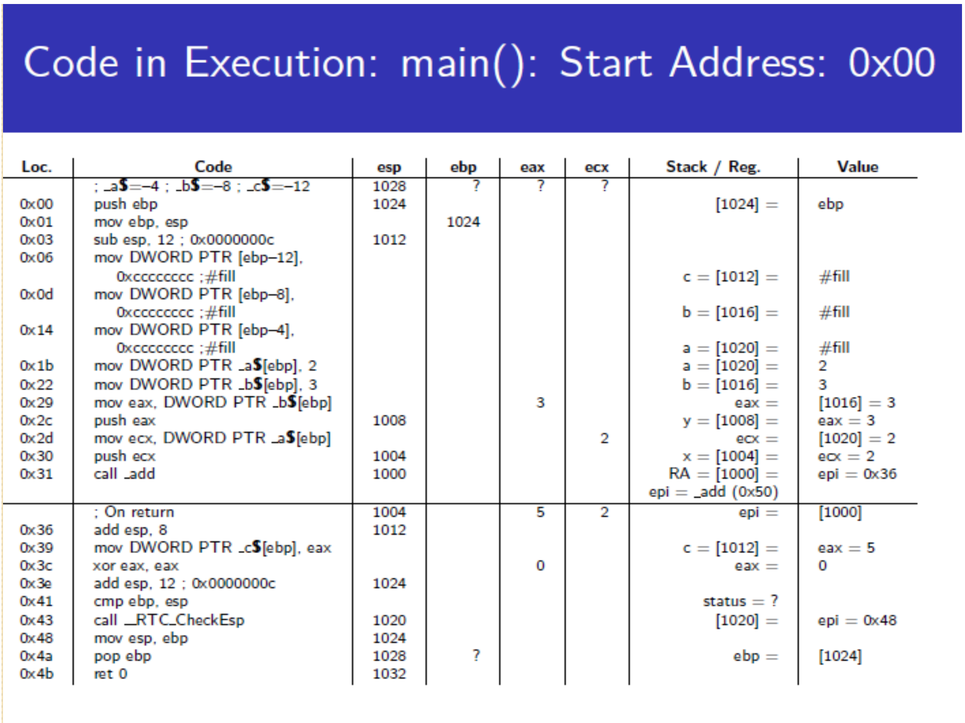
1012	−12	c
1016	−8	b = 3
1020	−4	a = 2
1024		ebp
1028		RA
1032	+8	argc
1036	+12	argv

ebp = 1024

AR of add()

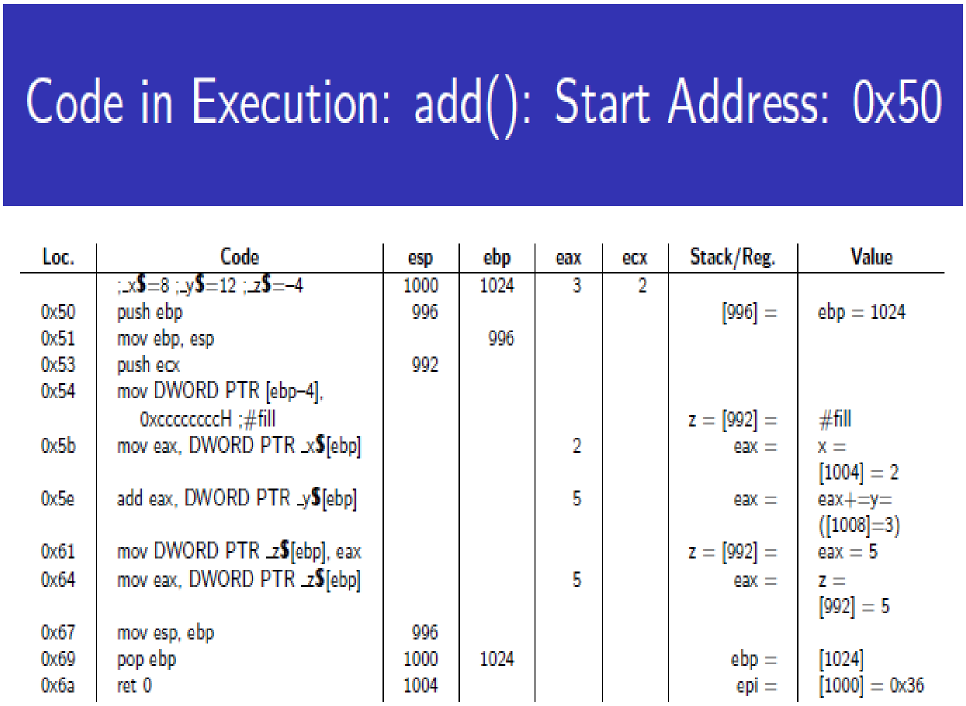
992	−4	z = 5
996		ebp = 1024
1000		RA
1004	+8	ecx = 2: x
1008	+12	eax = 3: y

ebp = 996



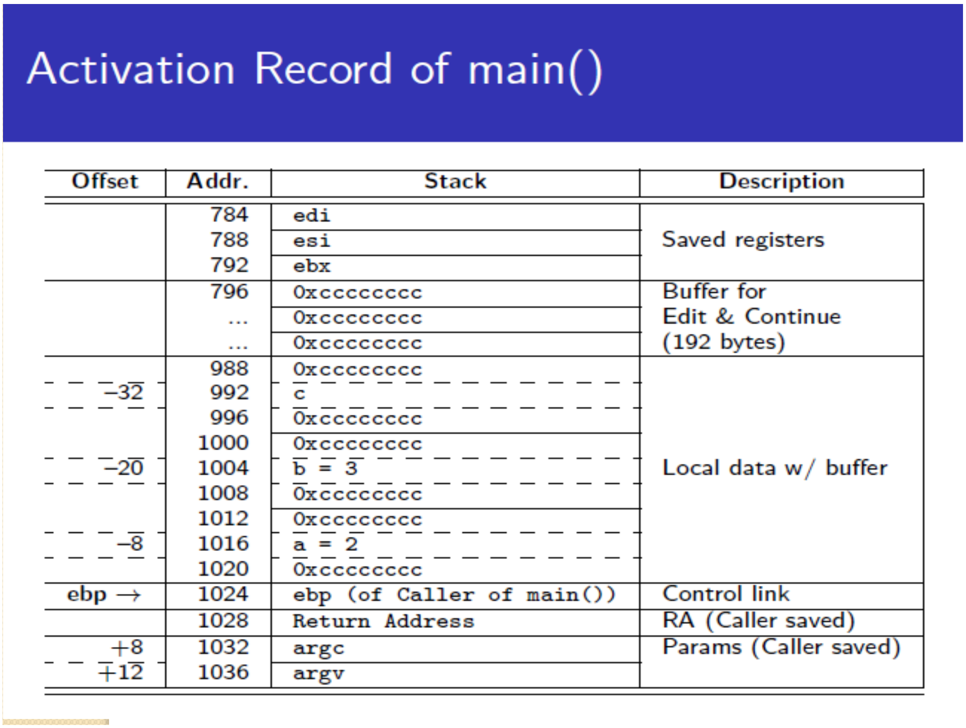
Code in Execution: main(): Start Address: 0x00

Loc.	Code	esp	ebp	eax	ecx	Stack / Reg.	Value
	; _a$=−4 ; _b$=−8 ; _c$=−12	1028	?	?	?		
0x00	push ebp	1024				[1024] =	ebp
0x01	mov ebp, esp		1024				
0x03	sub esp, 12 ; 0x0000000c	1012					
0x06	mov DWORD PTR [ebp−12],						
	0xcccccccc ;#fill					c = [1012] =	#fill
0x0d	mov DWORD PTR [ebp−8],						
	0xcccccccc ;#fill					b = [1016] =	#fill
0x14	mov DWORD PTR [ebp−4],						
	0xcccccccc ;#fill					a = [1020] =	#fill
0x1b	mov DWORD PTR _a$[ebp], 2					a = [1020] =	2
0x22	mov DWORD PTR _b$[ebp], 3					b = [1016] =	3
0x29	mov eax, DWORD PTR _b$[ebp]			3		eax =	[1016] = 3
0x2c	push eax	1008				y = [1008] =	eax = 3
0x2d	mov ecx, DWORD PTR _a$[ebp]				2	ecx =	[1020] = 2
0x30	push ecx	1004				x = [1004] =	ecx = 2
0x31	call _add	1000				RA = [1000] =	epi = 0x36
						epi = _add (0x50)	
	; On return	1004		5	2	epi =	[1000]
0x36	add esp, 8	1012					
0x39	mov DWORD PTR _c$[ebp], eax					c = [1012] =	eax = 5
0x3c	xor eax, eax			0		eax =	0
0x3e	add esp, 12 ; 0x0000000c	1024					
0x41	cmp ebp, esp					status = ?	
0x43	call __RTC_CheckEsp	1020				[1020] =	epi = 0x48
0x48	mov esp, ebp	1024					
0x4a	pop ebp	1028	?			ebp =	[1024]
0x4b	ret 0	1032					



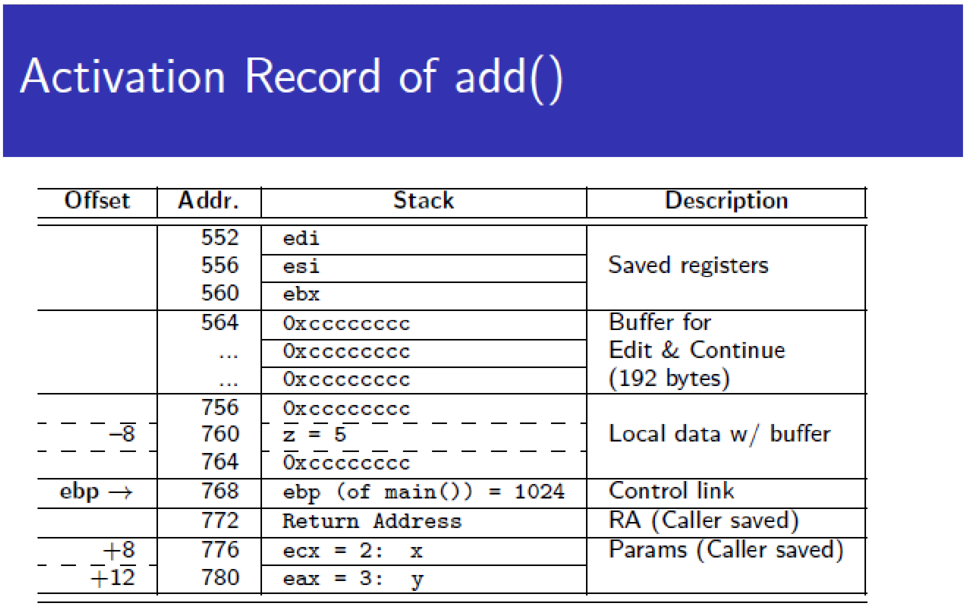
Code in Execution: add(): Start Address: 0x50

Loc.	Code	esp	ebp	eax	ecx	Stack/Reg.	Value
	;_x$=8 ;_y$=12 ;_z$=−4	1000	1024	3	2		
0x50	push ebp	996				[996] =	ebp = 1024
0x51	mov ebp, esp		996				
0x53	push ecx	992					
0x54	mov DWORD PTR [ebp−4], 0xccccccccH ;#fill					z = [992] =	#fill
0x5b	mov eax, DWORD PTR _x$[ebp]			2		eax =	x = [1004] = 2
0x5e	add eax, DWORD PTR _y$[ebp]			5		eax =	eax+=y= ([1008]=3)
0x61	mov DWORD PTR _z$[ebp], eax					z = [992] =	eax = 5
0x64	mov eax, DWORD PTR _z$[ebp]			5		eax =	z = [992] = 5
0x67	mov esp, ebp	996					
0x69	pop ebp	1000	1024			ebp =	[1024]
0x6a	ret 0	1004				epi =	[1000] = 0x36



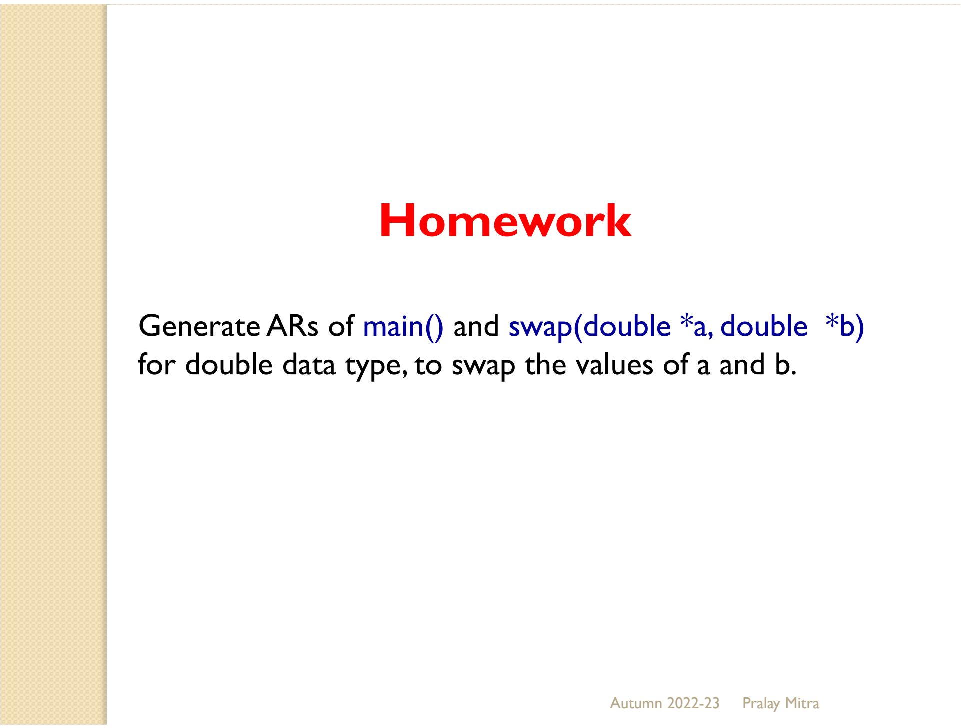
Activation Record of main()

Offset	Addr.	Stack	Description
	784	edi	Saved registers
	788	esi	
	792	ebx	
	796	0xcccccccc	Buffer for Edit & Continue (192 bytes)
	…	0xcccccccc	
	…	0xcccccccc	
	988	0xcccccccc	Local data w/ buffer
−32	992	c	
	996	0xcccccccc	
	1000	0xcccccccc	
−20	1004	b = 3	
	1008	0xcccccccc	
	1012	0xcccccccc	
−8	1016	a = 2	
	1020	0xcccccccc	
ebp →	1024	ebp (of Caller of main())	Control link
	1028	Return Address	RA (Caller saved)
+8	1032	argc	Params (Caller saved)
+12	1036	argv	



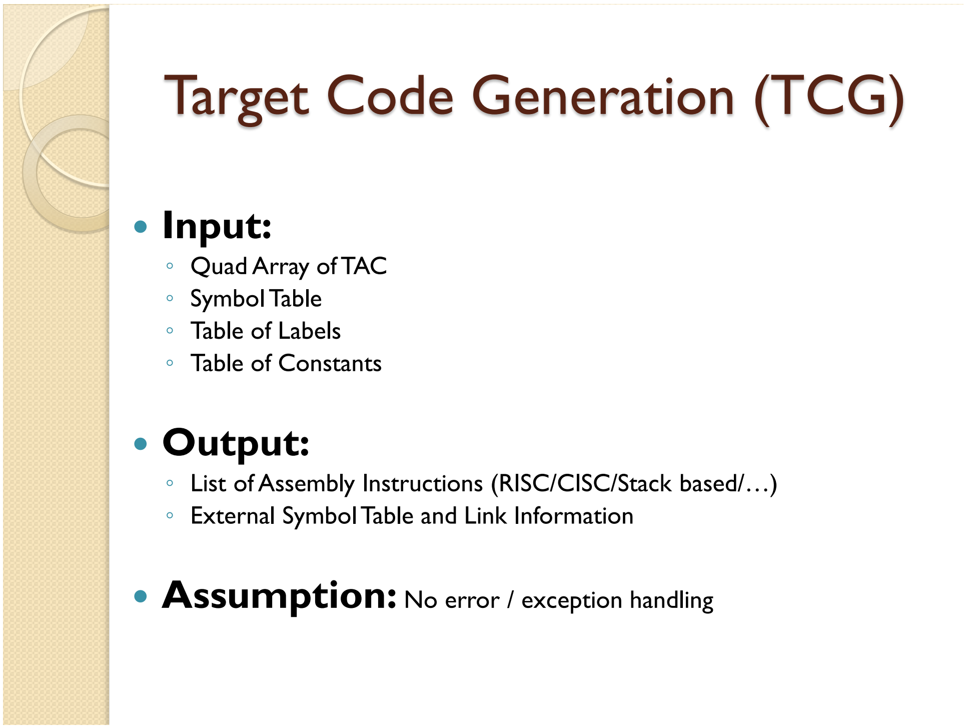
Activation Record of add()

Offset	Addr.	Stack	Description
	552	edi	Saved registers
	556	esi	
	560	ebx	
	564	0xcccccccc	Buffer for Edit & Continue (192 bytes)
	...	0xcccccccc	
	...	0xcccccccc	
	756	0xcccccccc	Local data w/ buffer
−8	760	z = 5	
	764	0xcccccccc	
ebp →	768	ebp (of main()) = 1024	Control link
	772	Return Address	RA (Caller saved)
+8	776	ecx = 2: x	Params (Caller saved)
+12	780	eax = 3: y	



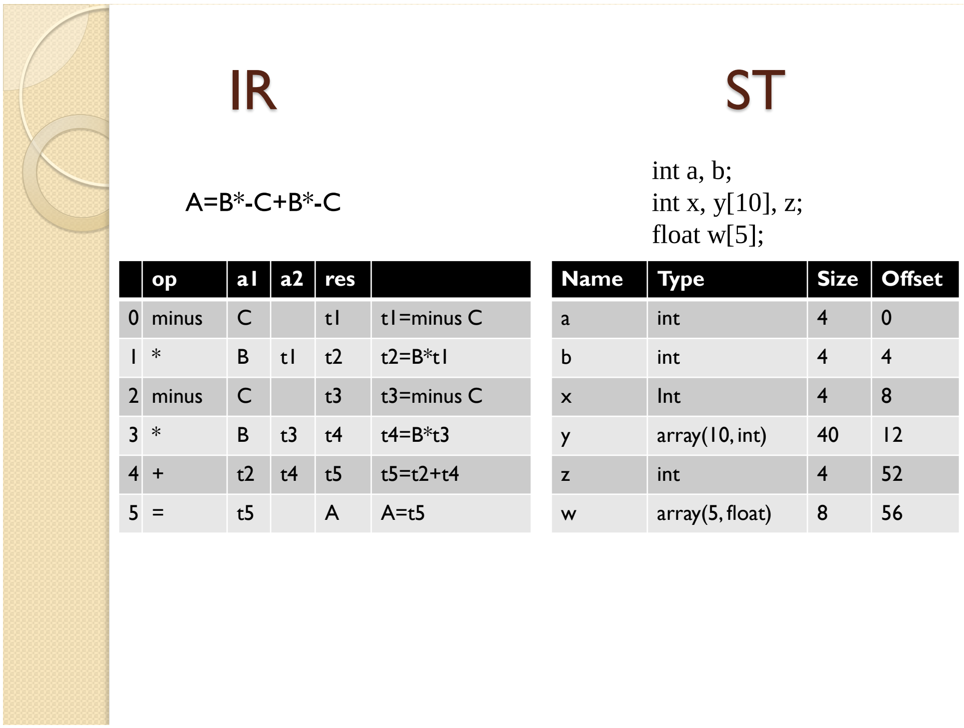
Homework

Generate ARs of main() and swap(double *a, double *b) for double data type, to swap the values of a and b.



Target Code Generation (TCG)

- **Input:**
  - Quad Array of TAC
  - Symbol Table
  - Table of Labels
  - Table of Constants
- **Output:**
  - List of Assembly Instructions (RISC/CISC/Stack based/…)
  - External Symbol Table and Link Information
- **Assumption:** No error / exception handling



IR

A=B*-C+B*-C

	op	a1	a2	res	
0	minus	C		t1	t1=minus C
1	*	B	t1	t2	t2=B*t1
2	minus	C		t3	t3=minus C
3	*	B	t3	t4	t4=B*t3
4	+	t2	t4	t5	t5=t2+t4
5	=	t5		A	A=t5

ST

int a, b;
int x, y[10], z;
float w[5];

Name	Type	Size	Offset
a	int	4	0
b	int	4	4
x	Int	4	8
y	array(10, int)	40	12
z	int	4	52
w	array(5, float)	8	56

ST

int fact(int i, double d);

ST(global) Symbol Table for global symbols

Name	Type	Init Val	Siz e	Offset	Nested Table
…	…	…	…	…	…
fact	function	null	0	…	Ptr-to-ST(fact)
…	…	…	…	…	…

ST(fact) Symbol Table for functon fact

Name	Type	Init Val	Siz e	Offset	Nested Table
i	int	null	4	0	null
d	double	null	8	4	null
retval	int	null	4	12	null

TCG - tasks

- **Instruction Selection**
  - Level of the IR
    - High level IR ---- code templates
    - Low level IR --- direct translation
  - Nature of the instruction set architecture
  - Desired quality of the generated code

x = y + z

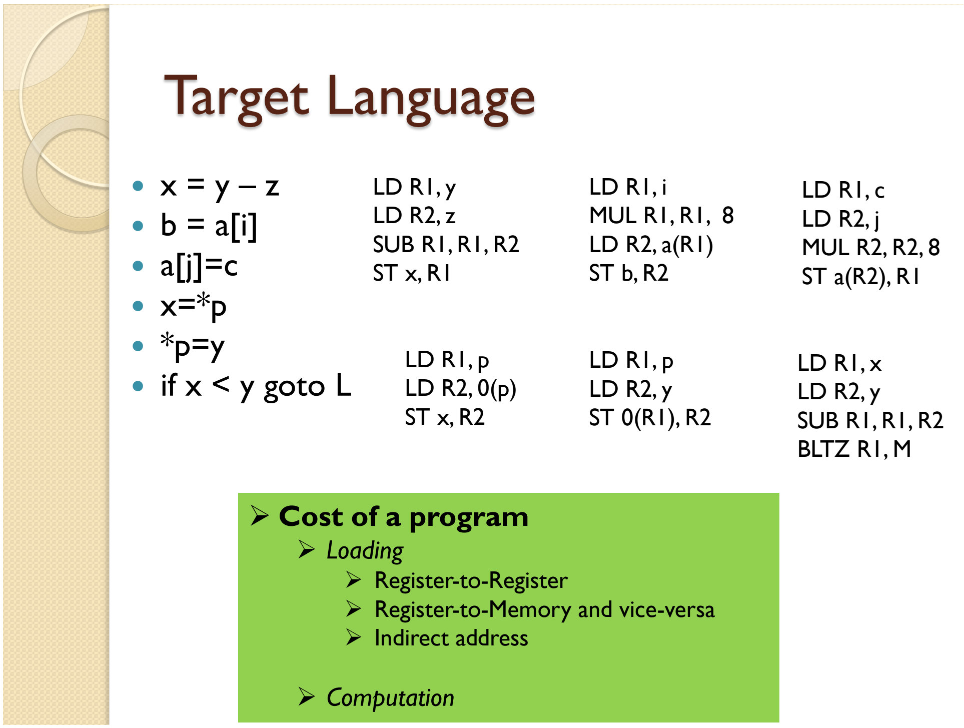
```
LD R0, y        // load y into register R0
ADD R0, R0, z   // add z to R0
ST  x, R0       // store R0 into x
```

Now,

a = b + c
d = a + e
a = a + 1   // INC a

- **Register Allocation and Assignment**
  - Allocation
  - Assignment

- **Instruction Ordering**



Target Language

- x = y – z
- b = a[i]
- a[j]=c
- x=*p
- *p=y
- if x < y goto L

```
LD R1, y
LD R2, z
SUB R1, R1, R2
ST x, R1
```

```
LD R1, i
MUL R1, R1,  8
LD R2, a(R1)
ST b, R2
```

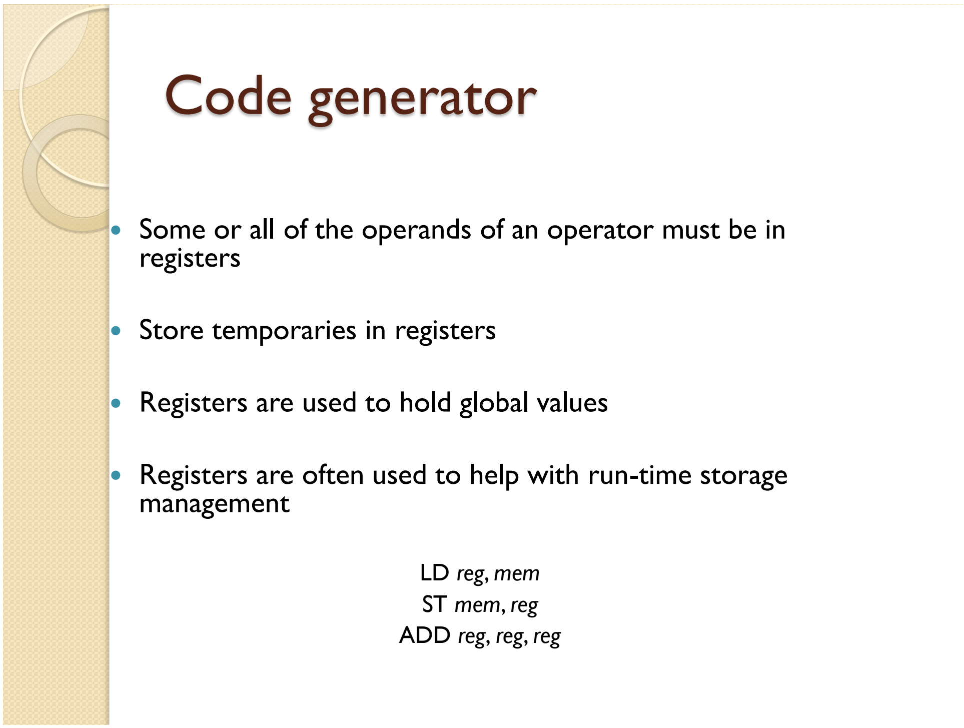
```
LD R1, c
LD R2, j
MUL R2, R2, 8
ST a(R2), R1
```

```
LD R1, p
LD R2, 0(p)
ST x, R2
```

```
LD R1, p
LD R2, y
ST 0(R1), R2
```

```
LD R1, x
LD R2, y
SUB R1, R1, R2
BLTZ R1, M
```

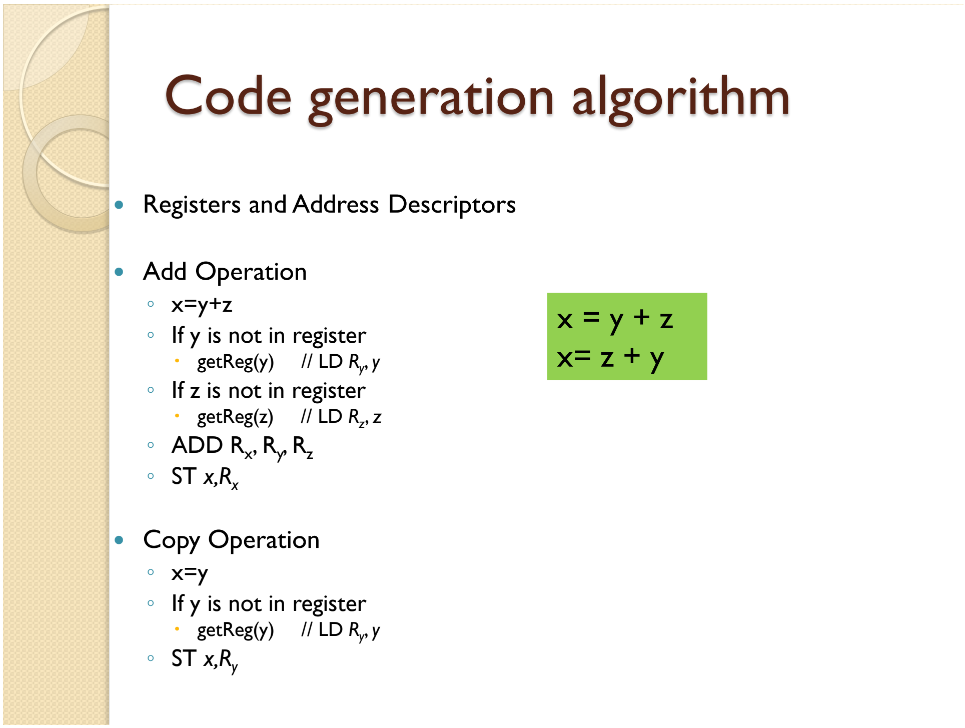
- **Cost of a program**
  - *Loading*
    - Register-to-Register
    - Register-to-Memory and vice-versa
    - Indirect address
  - *Computation*



Code generator

- Some or all of the operands of an operator must be in registers
- Store temporaries in registers
- Registers are used to hold global values
- Registers are often used to help with run-time storage management

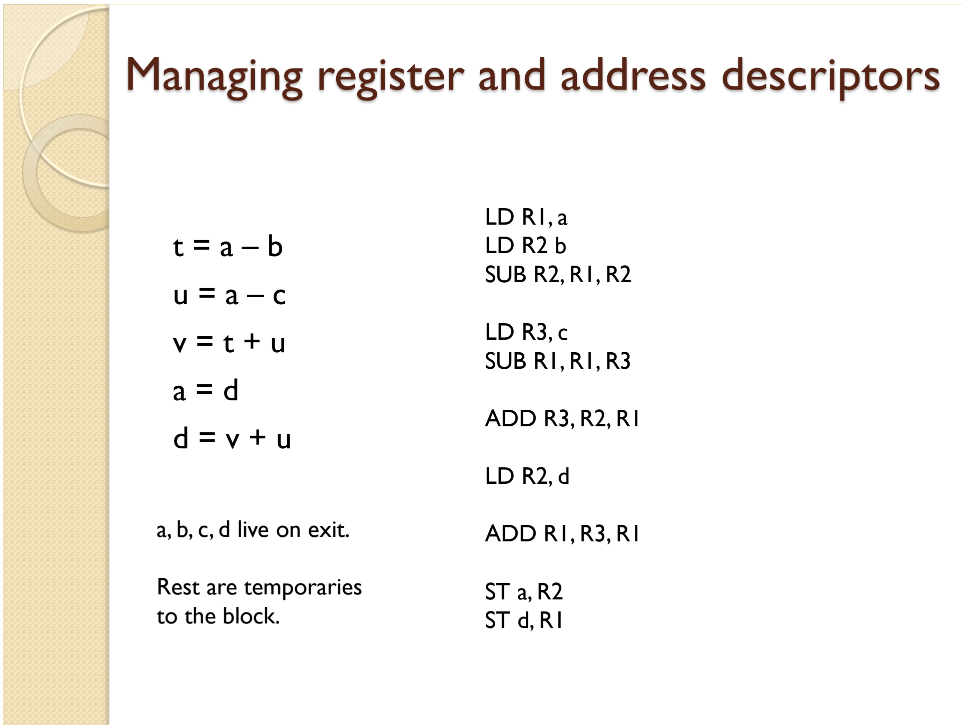
LD *reg*, *mem*
ST *mem*, *reg*
ADD *reg*, *reg*, *reg*



Code generation algorithm

- Registers and Address Descriptors
- Add Operation
  - x=y+z
  - If y is not in register
    - getReg(y) // LD R_y, *y*
  - If z is not in register
    - getReg(z) // LD R_z, *z*
  - ADD R_x, R_y, R_z
  - ST *x*,R_x
- Copy Operation
  - x=y
  - If y is not in register
    - getReg(y) // LD R_y, *y*
  - ST *x*,R_y

x = y + z
x= z + y



Managing register and address descriptors

t = a – b

u = a – c

v = t + u

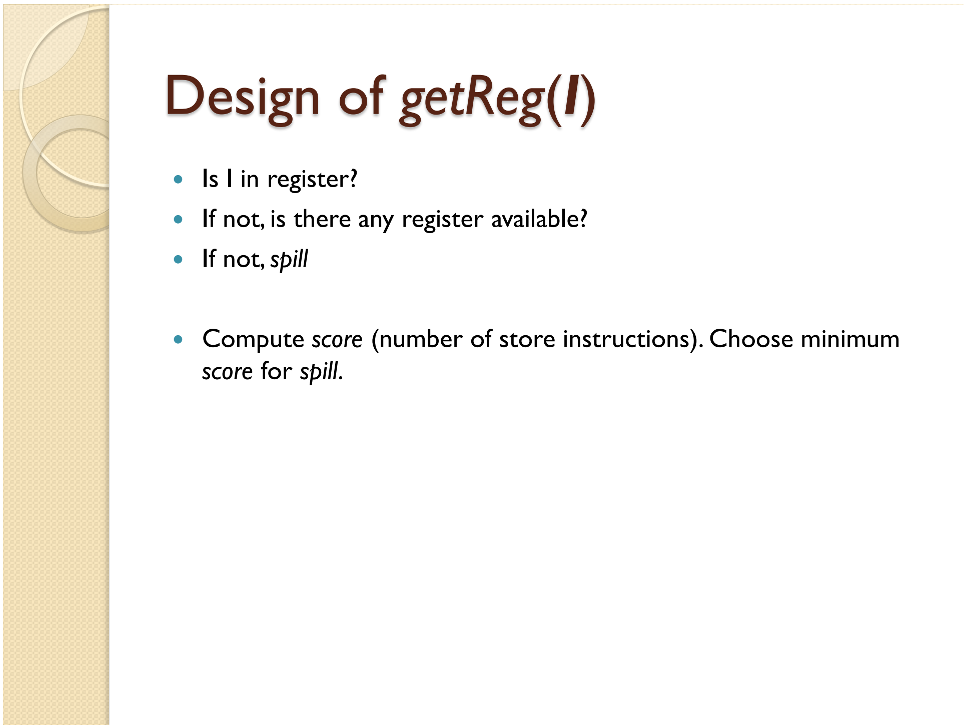
a = d

d = v + u

a, b, c, d live on exit.

Rest are temporaries
to the block.

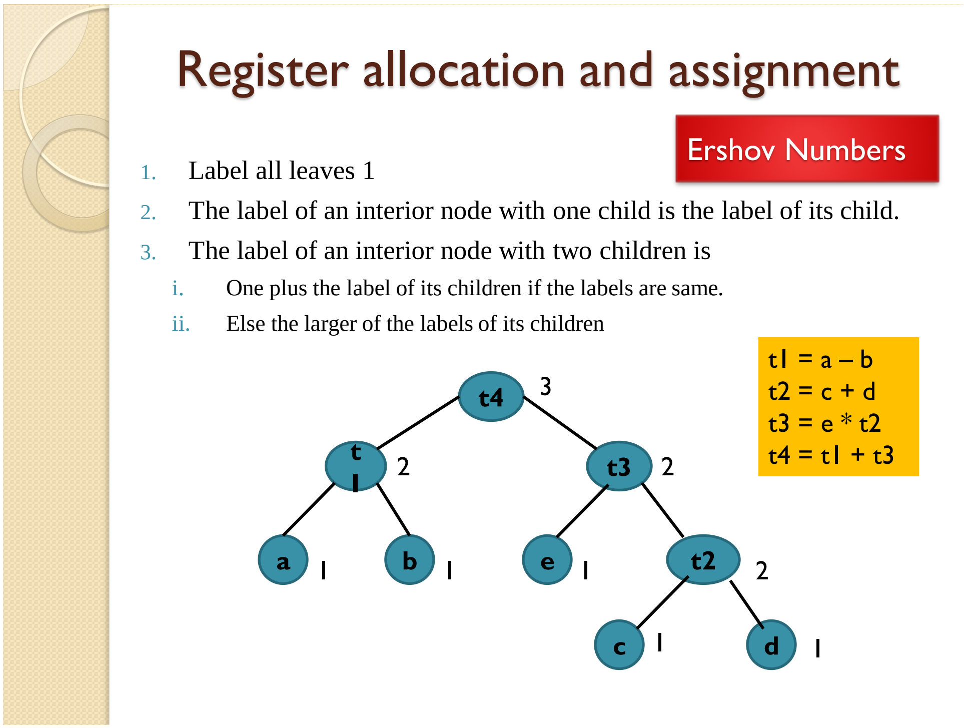
```
LD R1, a
LD R2 b
SUB R2, R1, R2

LD R3, c
SUB R1, R1, R3

ADD R3, R2, R1

LD R2, d

ADD R1, R3, R1

ST a, R2
ST d, R1
```



Design of *getReg(I)*

- Is I in register?
- If not, is there any register available?
- If not, *spill*

- Compute *score* (number of store instructions). Choose minimum *score* for *spill*.

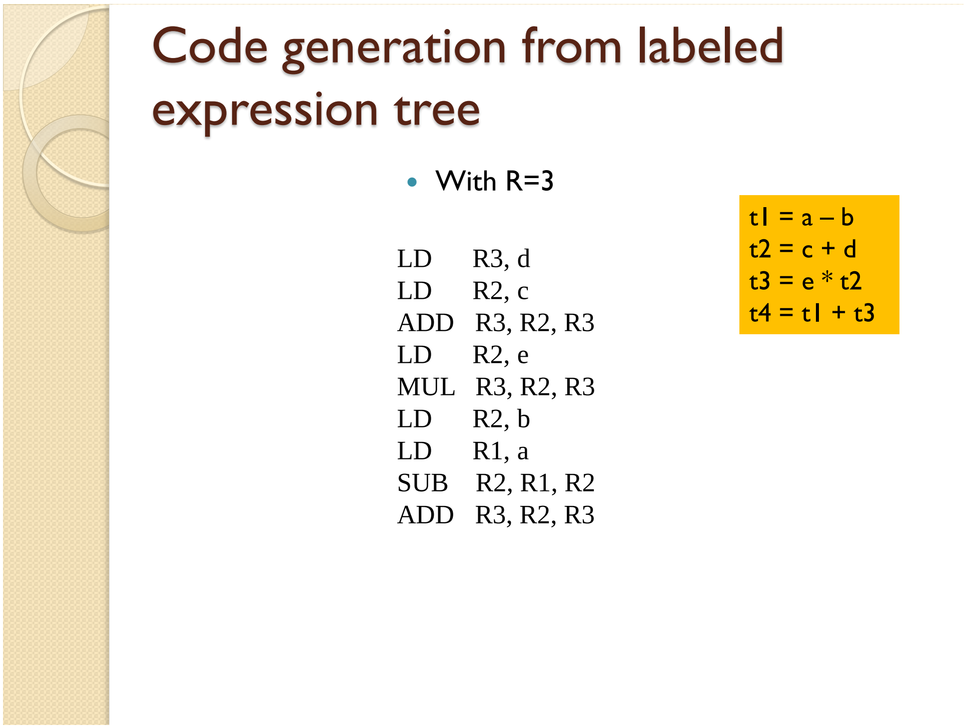


Register allocation and assignment

Ershov Numbers

1. Label all leaves 1
2. The label of an interior node with one child is the label of its child.
3. The label of an interior node with two children is
   i. One plus the label of its children if the labels are same.
   ii. Else the larger of the labels of its children

t1 = a – b
t2 = c + d
t3 = e * t2
t4 = t1 + t3

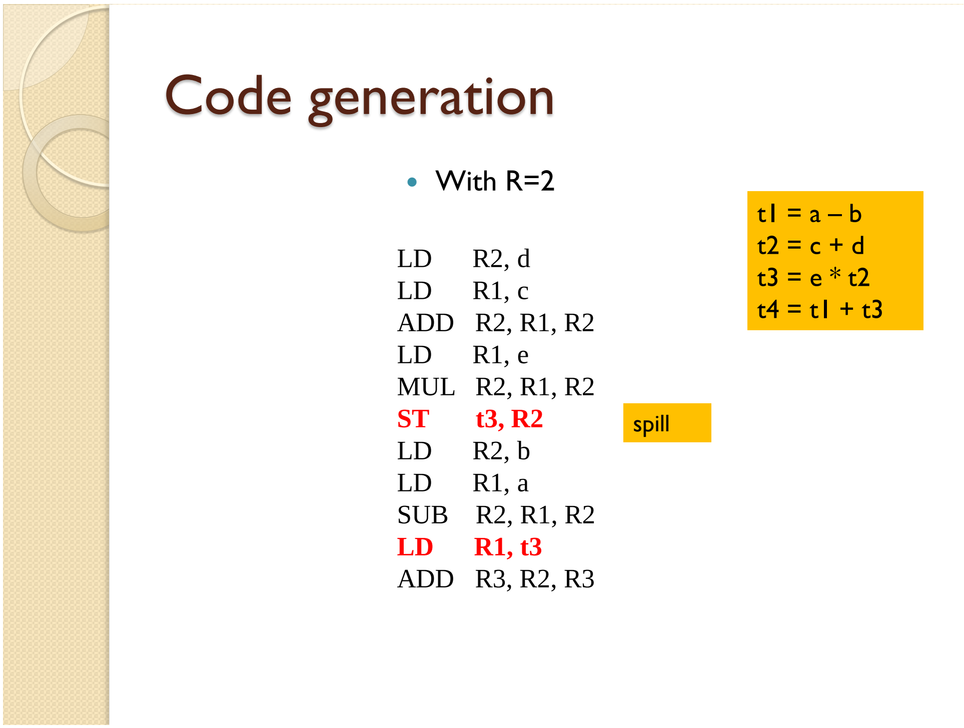


Code generation from labeled expression tree

- With R=3

```
LD     R3, d
LD     R2, c
ADD    R3, R2, R3
LD     R2, e
MUL    R3, R2, R3
LD     R2, b
LD     R1, a
SUB    R2, R1, R2
ADD    R3, R2, R3
```

t1 = a – b
t2 = c + d
t3 = e * t2
t4 = t1 + t3



Code generation

- With R=2

```
LD     R2, d
LD     R1, c
ADD    R2, R1, R2
LD     R1, e
MUL    R2, R1, R2
ST     t3, R2        spill
LD     R2, b
LD     R1, a
SUB    R2, R1, R2
LD     R1, t3
ADD    R3, R2, R3
```

t1 = a – b
t2 = c + d
t3 = e * t2
t4 = t1 + t3

TCG

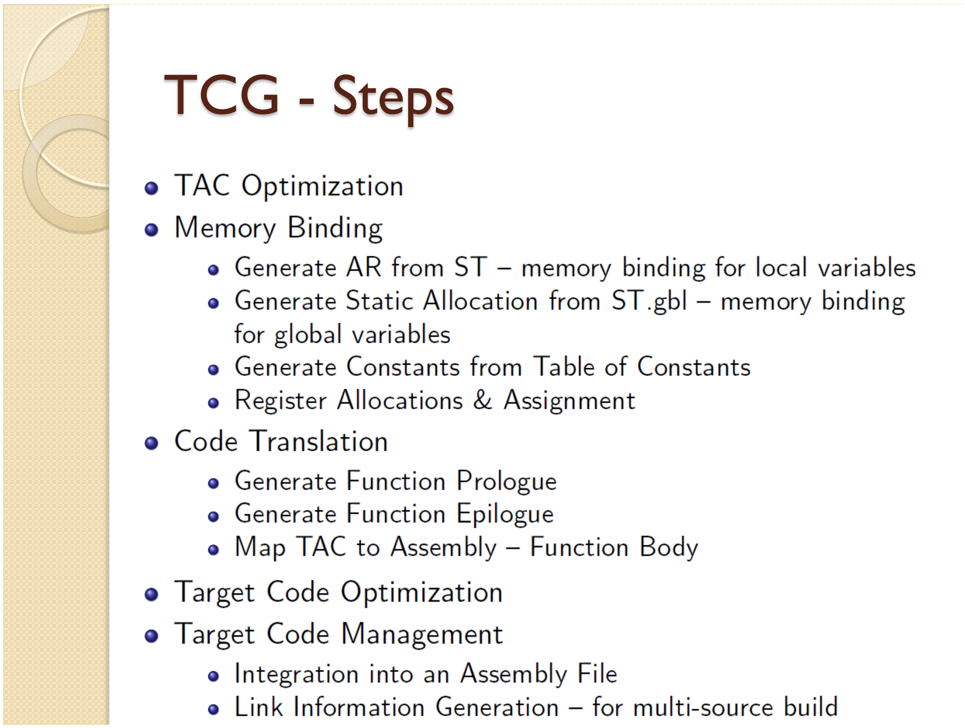
- Target Machine: `x86 32 bits`
- Input
  - Symbol Tables
  - Table of Labels
  - Table of Constants
  - Quad Array of TAC
- Output
  - List of Assembly Instructions
  - External Symbol Table and Link Information
- No Error / Exception Handling

Compilers (CS30003)

Lecture 27

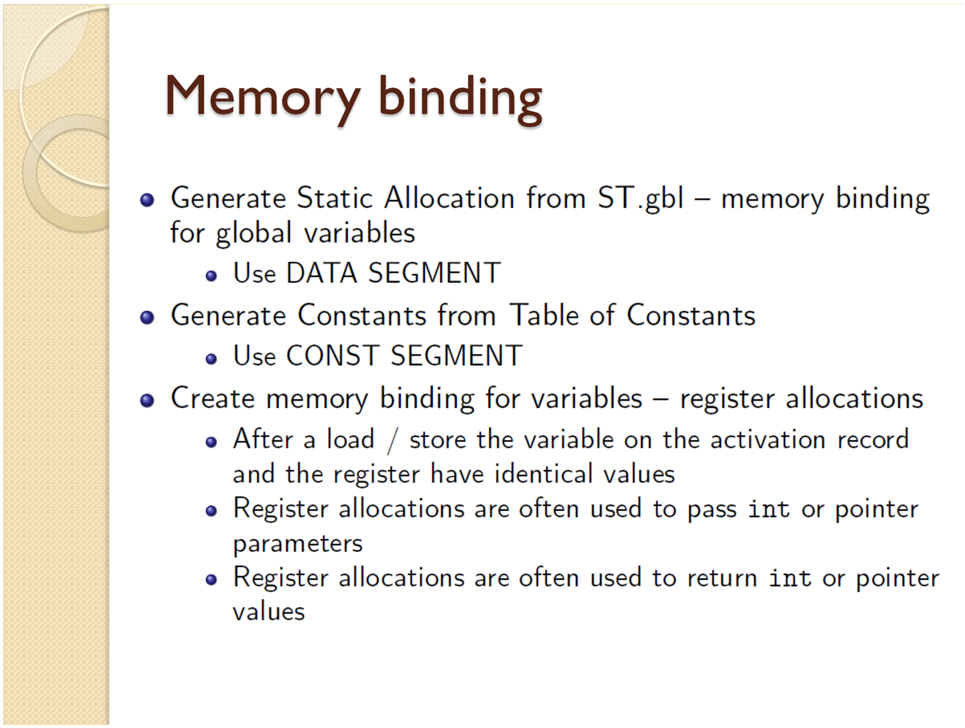
Pralay Mitra

Autumn 2023-24



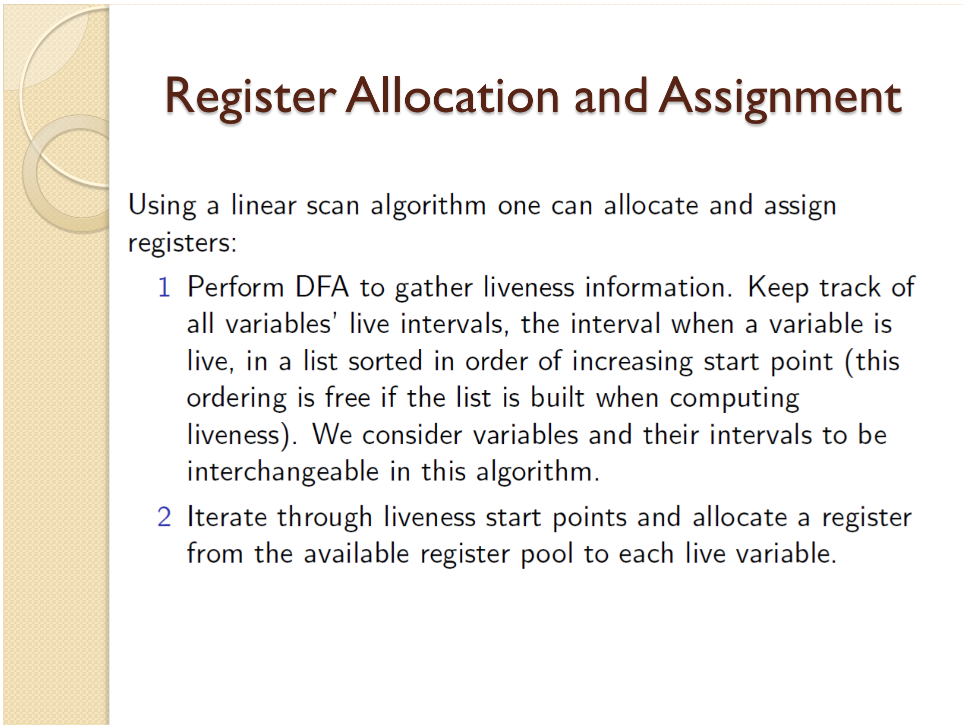
TCG - Steps

- TAC Optimization
- Memory Binding
  - Generate AR from ST – memory binding for local variables
  - Generate Static Allocation from ST.gbl – memory binding for global variables
  - Generate Constants from Table of Constants
  - Register Allocations & Assignment
- Code Translation
  - Generate Function Prologue
  - Generate Function Epilogue
  - Map TAC to Assembly – Function Body
- Target Code Optimization
- Target Code Management
  - Integration into an Assembly File
  - Link Information Generation – for multi-source build



Memory binding

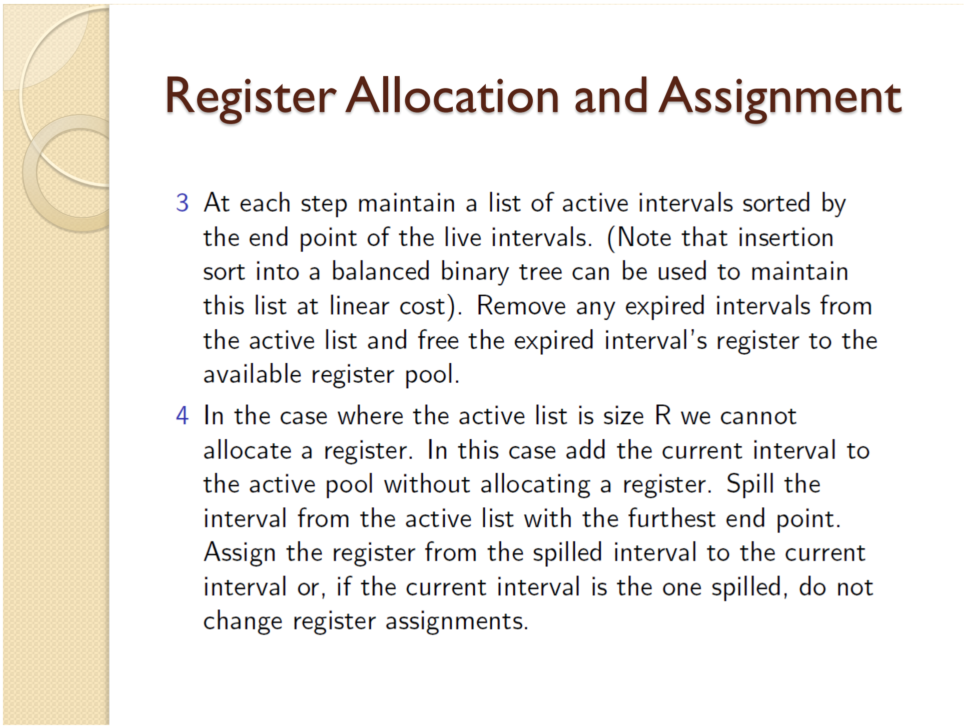
- Generate Static Allocation from ST.gbl – memory binding for global variables
  - Use DATA SEGMENT
- Generate Constants from Table of Constants
  - Use CONST SEGMENT
- Create memory binding for variables – register allocations
  - After a load / store the variable on the activation record and the register have identical values
  - Register allocations are often used to pass `int` or pointer parameters
  - Register allocations are often used to return `int` or pointer values



Register Allocation and Assignment

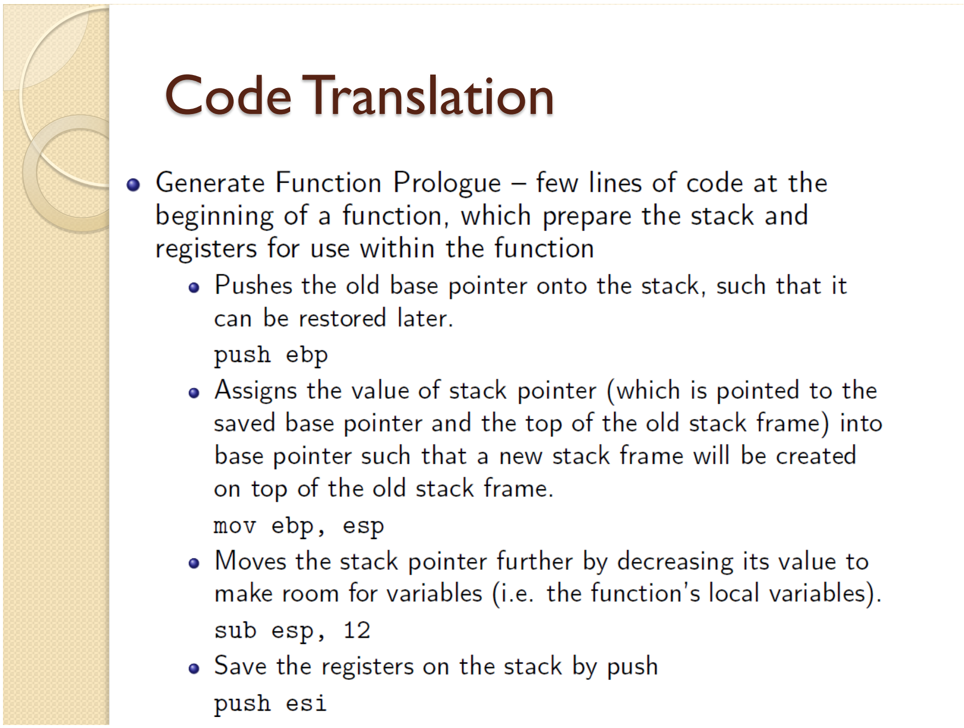
Using a linear scan algorithm one can allocate and assign registers:

1. Perform DFA to gather liveness information. Keep track of all variables' live intervals, the interval when a variable is live, in a list sorted in order of increasing start point (this ordering is free if the list is built when computing liveness). We consider variables and their intervals to be interchangeable in this algorithm.
2. Iterate through liveness start points and allocate a register from the available register pool to each live variable.



Register Allocation and Assignment

3. At each step maintain a list of active intervals sorted by the end point of the live intervals. (Note that insertion sort into a balanced binary tree can be used to maintain this list at linear cost). Remove any expired intervals from the active list and free the expired interval's register to the available register pool.
4. In the case where the active list is size R we cannot allocate a register. In this case add the current interval to the active pool without allocating a register. Spill the interval from the active list with the furthest end point. Assign the register from the spilled interval to the current interval or, if the current interval is the one spilled, do not change register assignments.



Code Translation

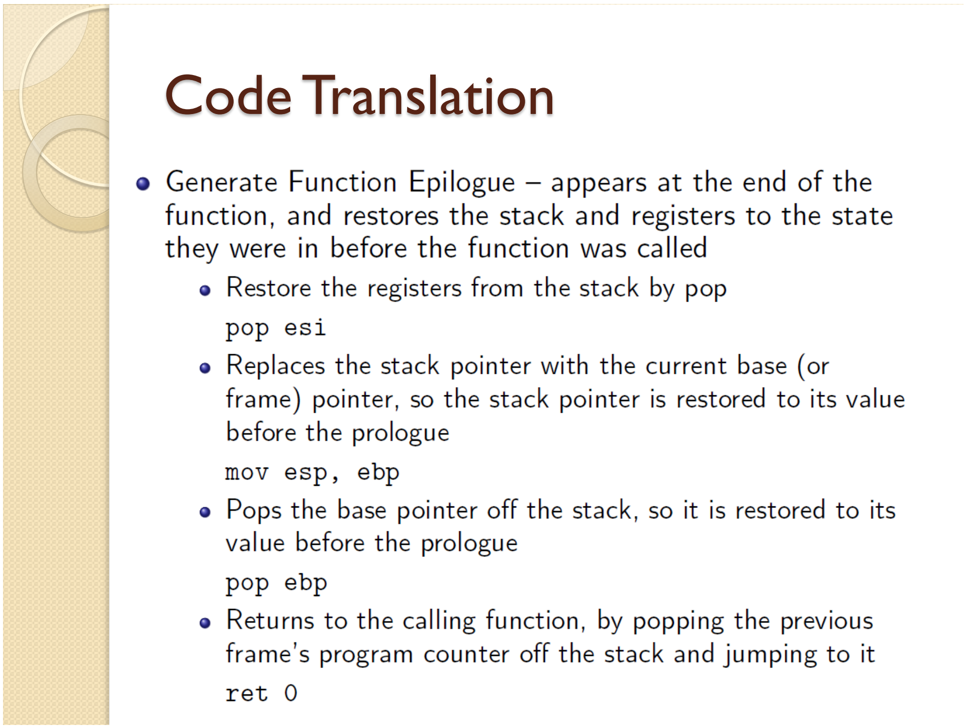
- Generate Function Prologue – few lines of code at the beginning of a function, which prepare the stack and registers for use within the function
  - Pushes the old base pointer onto the stack, such that it can be restored later.

    ```
    push ebp
    ```
  - Assigns the value of stack pointer (which is pointed to the saved base pointer and the top of the old stack frame) into base pointer such that a new stack frame will be created on top of the old stack frame.

    ```
    mov ebp, esp
    ```
  - Moves the stack pointer further by decreasing its value to make room for variables (i.e. the function's local variables).

    ```
    sub esp, 12
    ```
  - Save the registers on the stack by push

    ```
    push esi
    ```



Code Translation

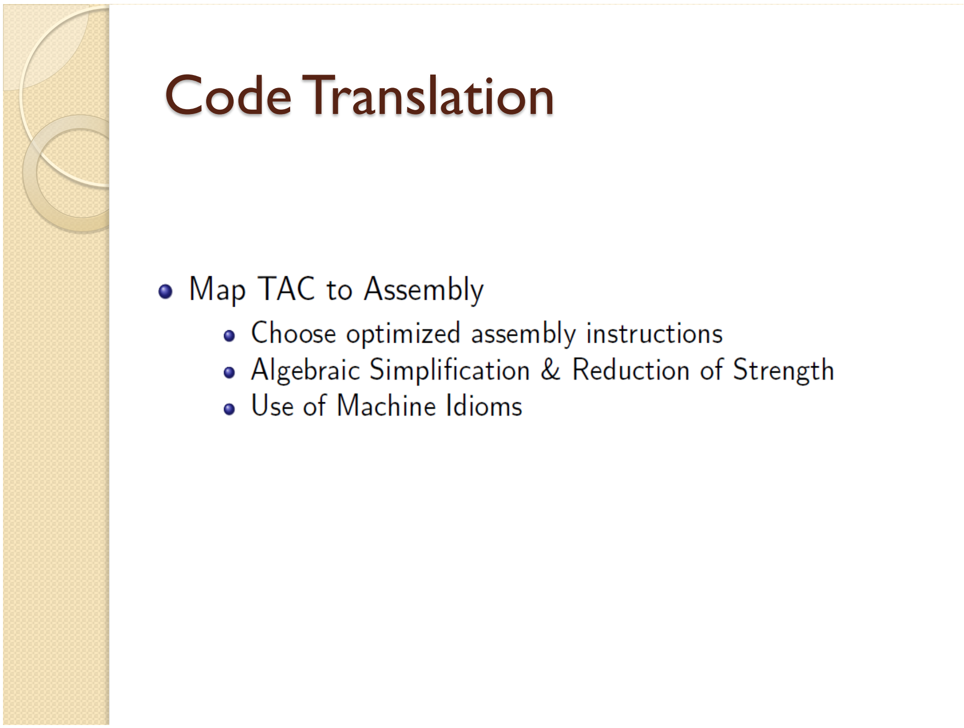
- Generate Function Epilogue – appears at the end of the function, and restores the stack and registers to the state they were in before the function was called
  - Restore the registers from the stack by pop

    ```
    pop esi
    ```
  - Replaces the stack pointer with the current base (or frame) pointer, so the stack pointer is restored to its value before the prologue

    ```
    mov esp, ebp
    ```
  - Pops the base pointer off the stack, so it is restored to its value before the prologue

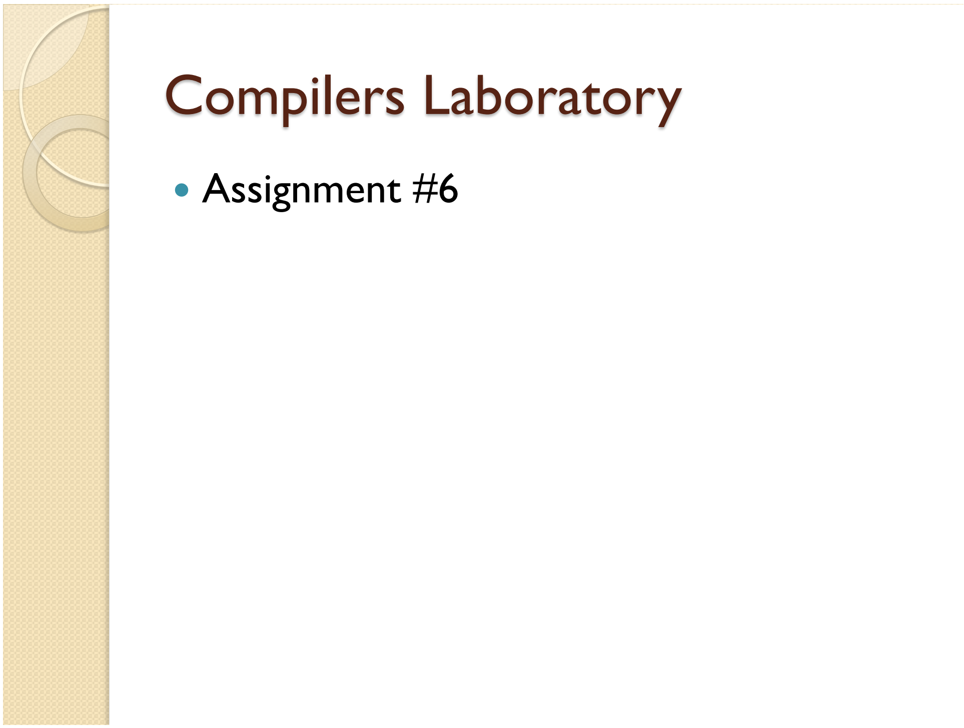
    ```
    pop ebp
    ```
  - Returns to the calling function, by popping the previous frame's program counter off the stack and jumping to it

    ```
    ret 0
    ```



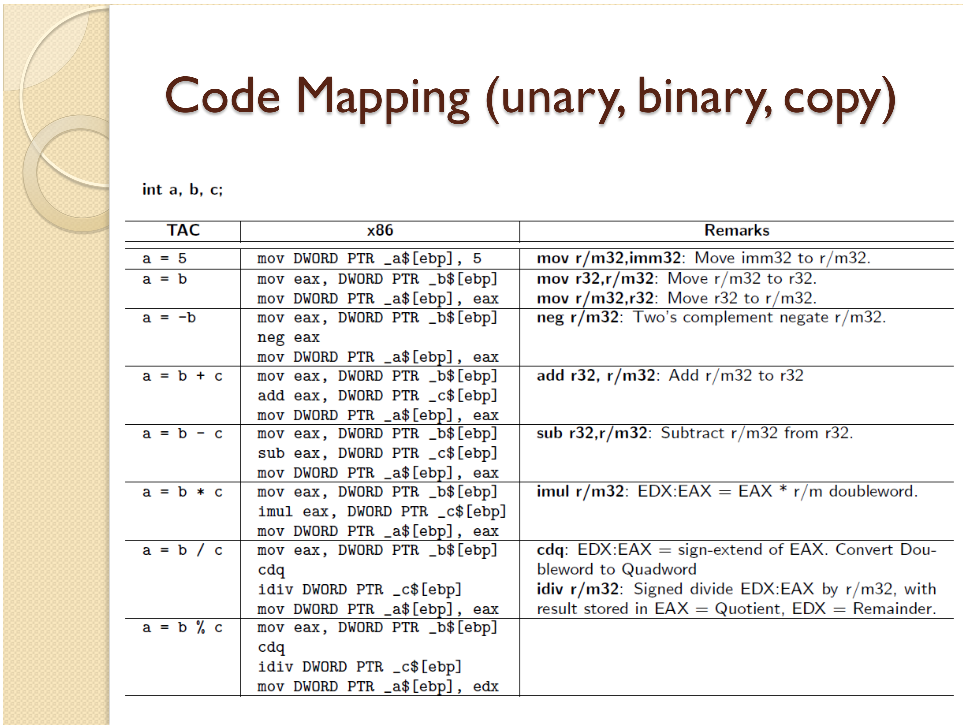
Code Translation

- Map TAC to Assembly
  - Choose optimized assembly instructions
  - Algebraic Simplification & Reduction of Strength
  - Use of Machine Idioms



Compilers Laboratory

- Assignment #6



Code Mapping (unary, binary, copy)

int a, b, c;

TAC	x86	Remarks
`a = 5`	`mov DWORD PTR _a$[ebp], 5`	**mov r/m32,imm32**: Move imm32 to r/m32.

`a = b`	`mov eax, DWORD PTR _b$[ebp]`
	`mov DWORD PTR _a$[ebp], eax`
		mov r32,r/m32: Move r/m32 to r32.
		mov r/m32,r32: Move r32 to r/m32.

`a = -b`	`mov eax, DWORD PTR _b$[ebp]`
	`neg eax`
	`mov DWORD PTR _a$[ebp], eax`
		neg r/m32: Two's complement negate r/m32.

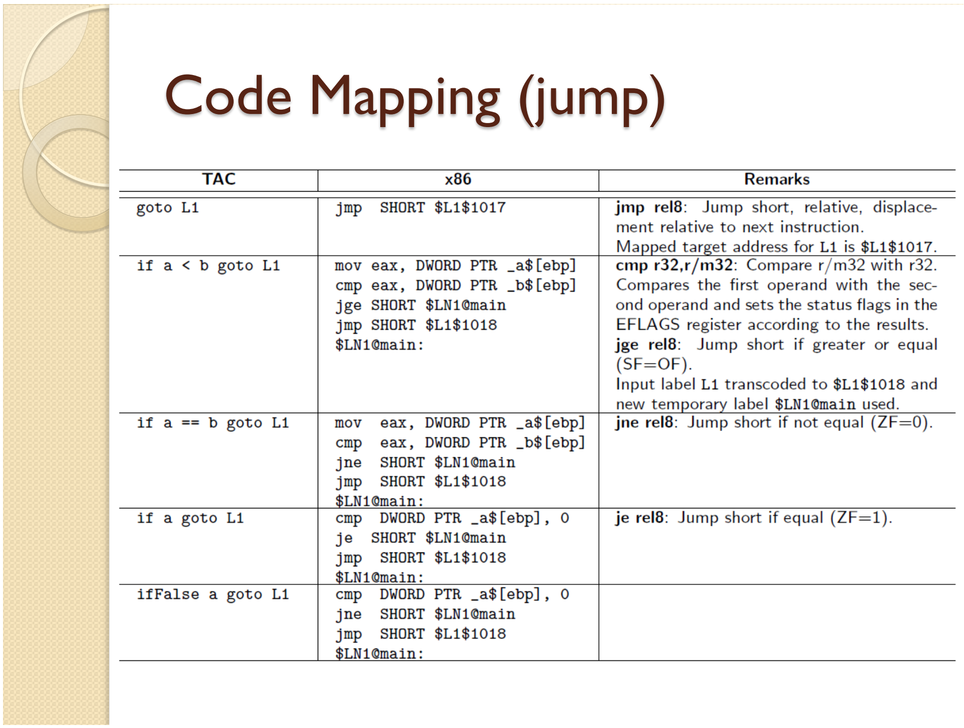
`a = b + c`	`mov eax, DWORD PTR _b$[ebp]`
	`add eax, DWORD PTR _c$[ebp]`
	`mov DWORD PTR _a$[ebp], eax`
		add r32, r/m32: Add r/m32 to r32

`a = b - c`	`mov eax, DWORD PTR _b$[ebp]`
	`sub eax, DWORD PTR _c$[ebp]`
	`mov DWORD PTR _a$[ebp], eax`
		sub r32,r/m32: Subtract r/m32 from r32.

`a = b * c`	`mov eax, DWORD PTR _b$[ebp]`
	`imul eax, DWORD PTR _c$[ebp]`
	`mov DWORD PTR _a$[ebp], eax`
		imul r/m32: EDX:EAX = EAX * r/m doubleword.

`a = b / c`	`mov eax, DWORD PTR _b$[ebp]`
	`cdq`
	`idiv DWORD PTR _c$[ebp]`
	`mov DWORD PTR _a$[ebp], eax`
		cdq: EDX:EAX = sign-extend of EAX. Convert Doubleword to Quadword
		idiv r/m32: Signed divide EDX:EAX by r/m32, with result stored in EAX = Quotient, EDX = Remainder.

`a = b % c`	`mov eax, DWORD PTR _b$[ebp]`
	`cdq`
	`idiv DWORD PTR _c$[ebp]`
	`mov DWORD PTR _a$[ebp], edx`	



Code Mapping (jump)

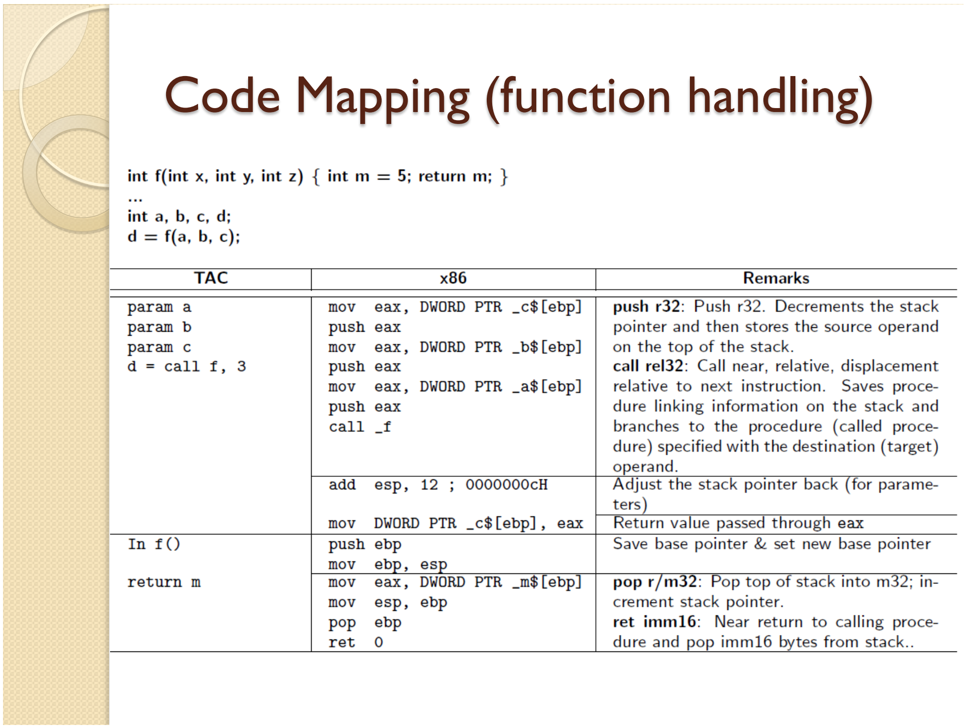
TAC	x86	Remarks
`goto L1`	`jmp SHORT $L1$1017`	**jmp rel8**: Jump short, relative, displacement relative to next instruction. Mapped target address for L1 is $L1$1017.

`if a < b goto L1`	`mov eax, DWORD PTR _a$[ebp]`
	`cmp eax, DWORD PTR _b$[ebp]`
	`jge SHORT $LN1@main`
	`jmp SHORT $L1$1018`
	`$LN1@main:`
		cmp r32,r/m32: Compare r/m32 with r32. Compares the first operand with the second operand and sets the status flags in the EFLAGS register according to the results.
		jge rel8: Jump short if greater or equal (SF=OF).
		Input label L1 transcoded to $L1$1018 and new temporary label `$LN1@main` used.

`if a == b goto L1`	`mov eax, DWORD PTR _a$[ebp]`
	`cmp eax, DWORD PTR _b$[ebp]`
	`jne SHORT $LN1@main`
	`jmp SHORT $L1$1018`
	`$LN1@main:`
		jne rel8: Jump short if not equal (ZF=0).

`if a goto L1`	`cmp DWORD PTR _a$[ebp], 0`
	`je SHORT $LN1@main`
	`jmp SHORT $L1$1018`
	`$LN1@main:`
		je rel8: Jump short if equal (ZF=1).

`ifFalse a goto L1`	`cmp DWORD PTR _a$[ebp], 0`
	`jne SHORT $LN1@main`
	`jmp SHORT $L1$1018`
	`$LN1@main:`	



Code Mapping (function handling)

int f(int x, int y, int z) { int m = 5; return m; }
...
int a, b, c, d;
d = f(a, b, c);

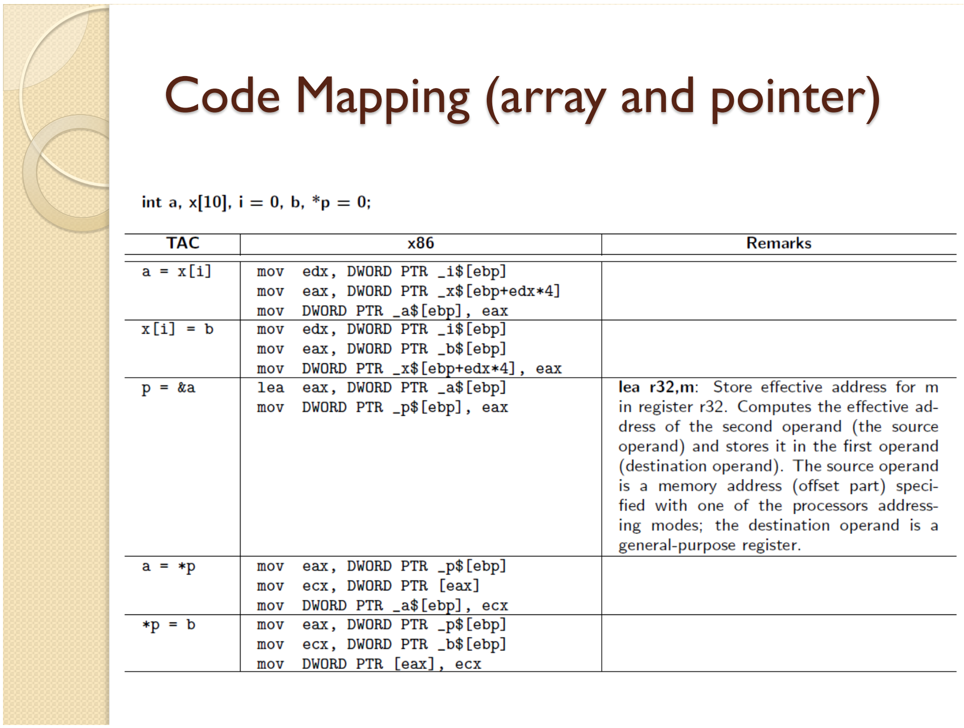
TAC	x86	Remarks

`param a`
`param b`
`param c`
`d = call f, 3`
	`mov eax, DWORD PTR _c$[ebp]`
	`push eax`
	`mov eax, DWORD PTR _b$[ebp]`
	`push eax`
	`mov eax, DWORD PTR _a$[ebp]`
	`push eax`
	`call _f`
		push r32: Push r32. Decrements the stack pointer and then stores the source operand on the top of the stack.
		call rel32: Call near, relative, displacement relative to next instruction. Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand.

	`add esp, 12 ; 0000000cH`	Adjust the stack pointer back (for parameters)
	`mov DWORD PTR _c$[ebp], eax`	Return value passed through `eax`

`In f()`	`push ebp`
	`mov ebp, esp`
		Save base pointer & set new base pointer

`return m`	`mov eax, DWORD PTR _m$[ebp]`
	`mov esp, ebp`
	`pop ebp`
	`ret 0`
		pop r/m32: Pop top of stack into m32; increment stack pointer.
		ret imm16: Near return to calling procedure and pop imm16 bytes from stack..



Code Mapping (array and pointer)

int a, x[10], i = 0, b, *p = 0;

TAC	x86	Remarks

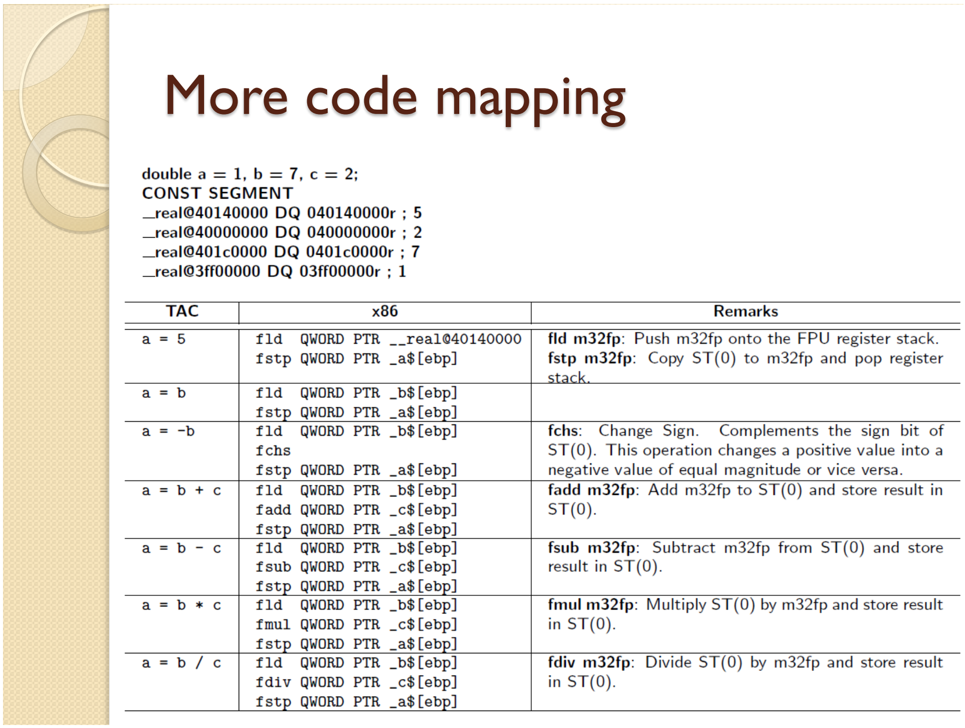
`a = x[i]`	`mov edx, DWORD PTR _i$[ebp]`
	`mov eax, DWORD PTR _x$[ebp+edx*4]`
	`mov DWORD PTR _a$[ebp], eax`	

`x[i] = b`	`mov edx, DWORD PTR _i$[ebp]`
	`mov eax, DWORD PTR _b$[ebp]`
	`mov DWORD PTR _x$[ebp+edx*4], eax`	

`p = &a`	`lea eax, DWORD PTR _a$[ebp]`
	`mov DWORD PTR _p$[ebp], eax`
		lea r32,m: Store effective address for m in register r32. Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register.

`a = *p`	`mov eax, DWORD PTR _p$[ebp]`
	`mov ecx, DWORD PTR [eax]`
	`mov DWORD PTR _a$[ebp], ecx`	

`*p = b`	`mov eax, DWORD PTR _p$[ebp]`
	`mov ecx, DWORD PTR _b$[ebp]`
	`mov DWORD PTR [eax], ecx`	



More code mapping

double a = 1, b = 7, c = 2;
CONST SEGMENT
__real@40140000 DQ 040140000r ; 5
__real@40000000 DQ 040000000r ; 2
__real@401c0000 DQ 0401c0000r ; 7
__real@3ff00000 DQ 03ff00000r ; 1

TAC	x86	Remarks

`a = 5`	`fld  QWORD PTR __real@40140000`
	`fstp QWORD PTR _a$[ebp]`
		fld m32fp: Push m32fp onto the FPU register stack.
		fstp m32fp: Copy ST(0) to m32fp and pop register stack.

`a = b`	`fld  QWORD PTR _b$[ebp]`
	`fstp QWORD PTR _a$[ebp]`	

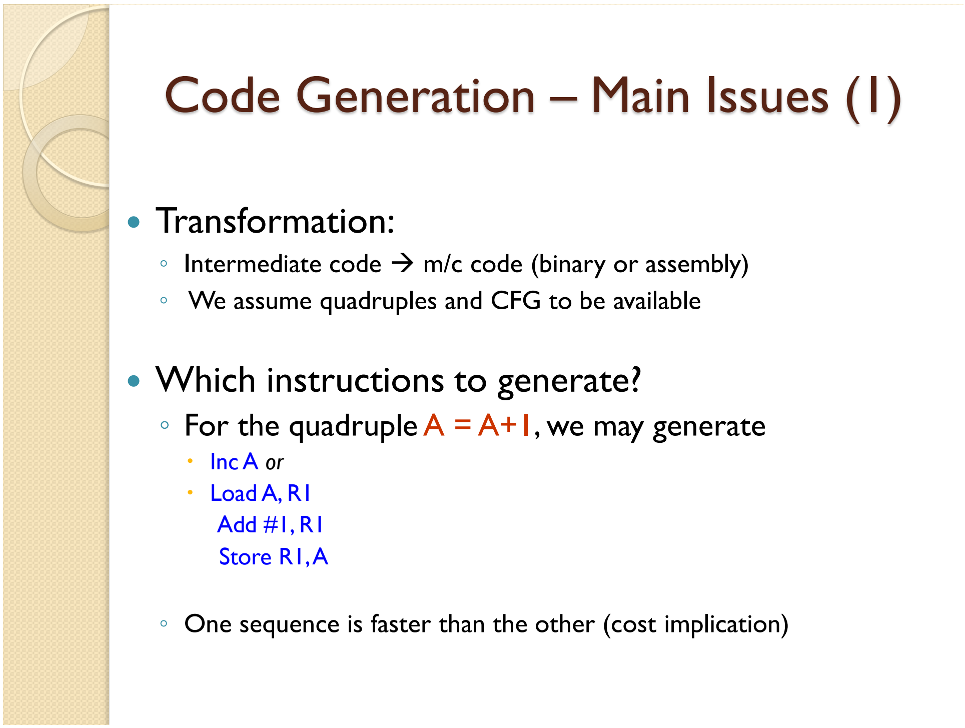
`a = -b`	`fld  QWORD PTR _b$[ebp]`
	`fchs`
	`fstp QWORD PTR _a$[ebp]`
		fchs: Change Sign. Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa.

`a = b + c`	`fld  QWORD PTR _b$[ebp]`
	`fadd QWORD PTR _c$[ebp]`
	`fstp QWORD PTR _a$[ebp]`
		fadd m32fp: Add m32fp to ST(0) and store result in ST(0).

`a = b - c`	`fld  QWORD PTR _b$[ebp]`
	`fsub QWORD PTR _c$[ebp]`
	`fstp QWORD PTR _a$[ebp]`
		fsub m32fp: Subtract m32fp from ST(0) and store result in ST(0).

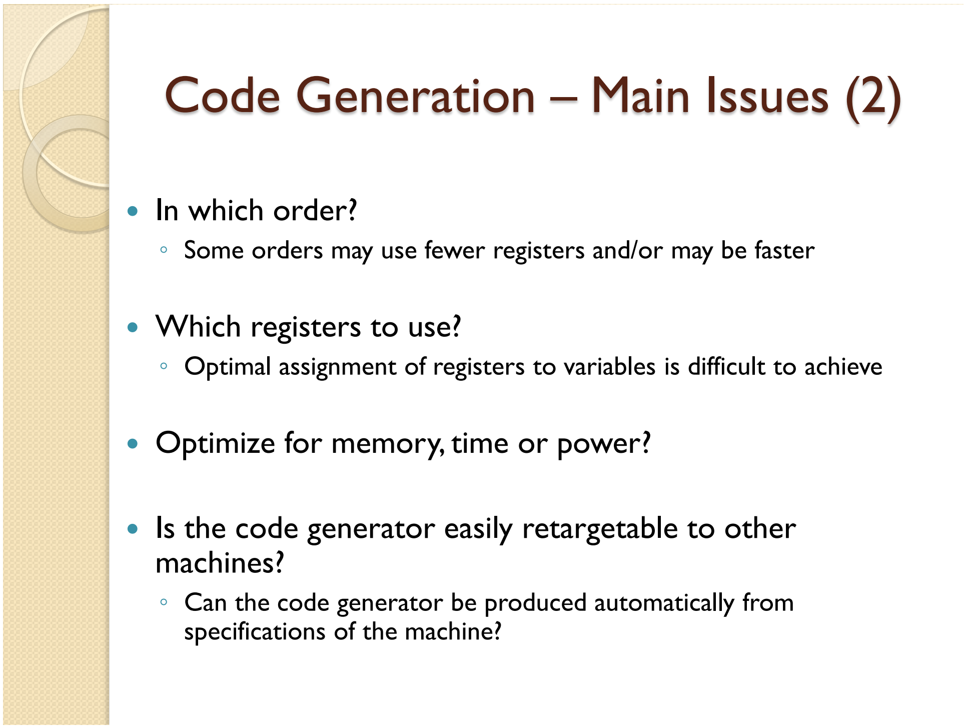
`a = b * c`	`fld  QWORD PTR _b$[ebp]`
	`fmul QWORD PTR _c$[ebp]`
	`fstp QWORD PTR _a$[ebp]`
		fmul m32fp: Multiply ST(0) by m32fp and store result in ST(0).

`a = b / c`	`fld  QWORD PTR _b$[ebp]`
	`fdiv QWORD PTR _c$[ebp]`
	`fstp QWORD PTR _a$[ebp]`
		fdiv m32fp: Divide ST(0) by m32fp and store result in ST(0).



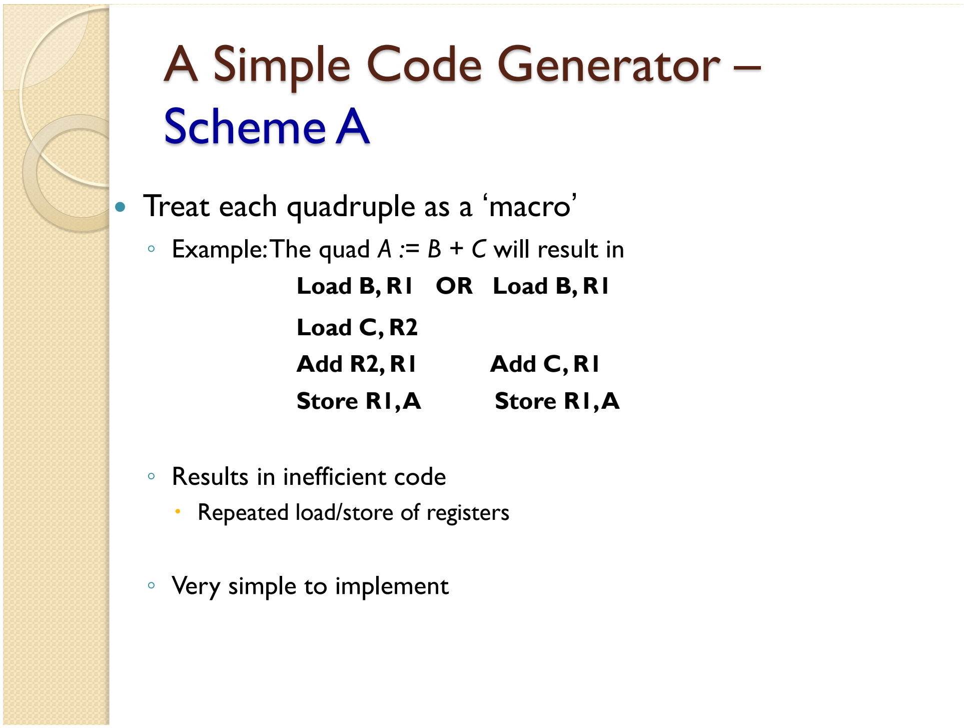
Code Generation – Main Issues (I)

- Transformation:
  - Intermediate code → m/c code (binary or assembly)
  - We assume quadruples and CFG to be available
- Which instructions to generate?
  - For the quadruple A = A+1, we may generate
    - Inc A *or*
    - Load A, R1
      Add #1, R1
      Store R1, A
  - One sequence is faster than the other (cost implication)



Code Generation – Main Issues (2)

- In which order?
  - Some orders may use fewer registers and/or may be faster
- Which registers to use?
  - Optimal assignment of registers to variables is difficult to achieve
- Optimize for memory, time or power?
- Is the code generator easily retargetable to other machines?
  - Can the code generator be produced automatically from specifications of the machine?



A Simple Code Generator – Scheme A

- Treat each quadruple as a ‘macro’
  - Example: The quad *A* := *B* + *C* will result in

```
Load B, R1    OR   Load B, R1
Load C, R2
Add R2, R1         Add C, R1
Store R1,A         Store R1,A
```

  - Results in inefficient code
    - Repeated load/store of registers
  - Very simple to implement

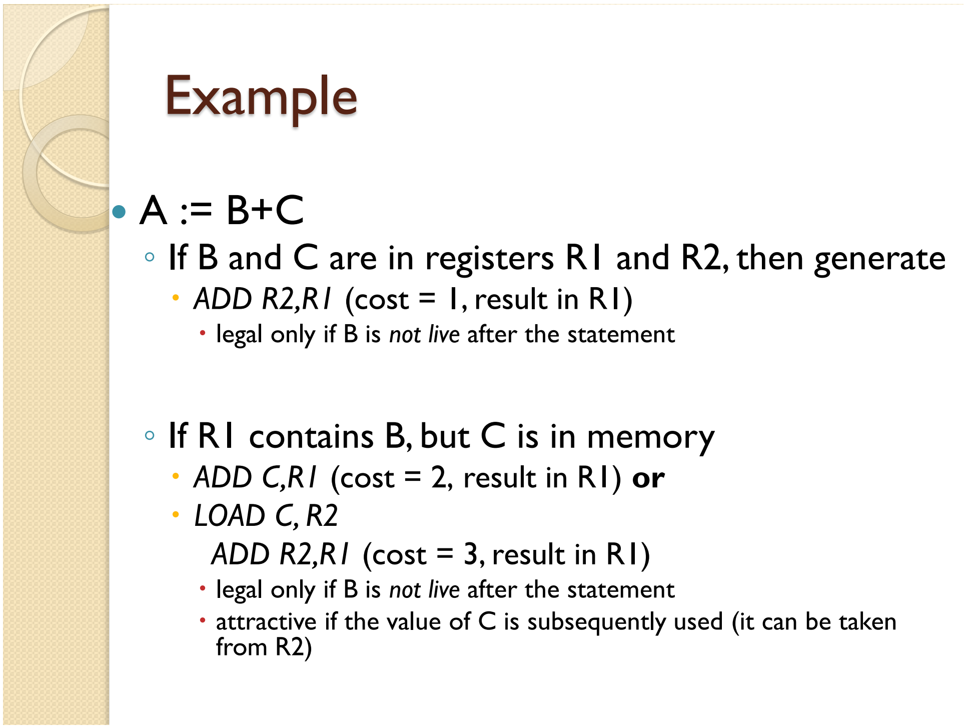
A Simple Code Generator – Scheme B

- Track values in registers and reuse them
  - If any operand is already in a register, take advantage of it
  - Register descriptors
    - Tracks <register, variable name> pairs
    - A single register can contain values of multiple names, if they are all copies
  - Address descriptors
    - Tracks <variable name, location> pairs
    - A single name may have its value in multiple locations, such as, memory, register, and stack

A Simple Code Generator – Scheme B

- Leave computed result in a register as long as possible

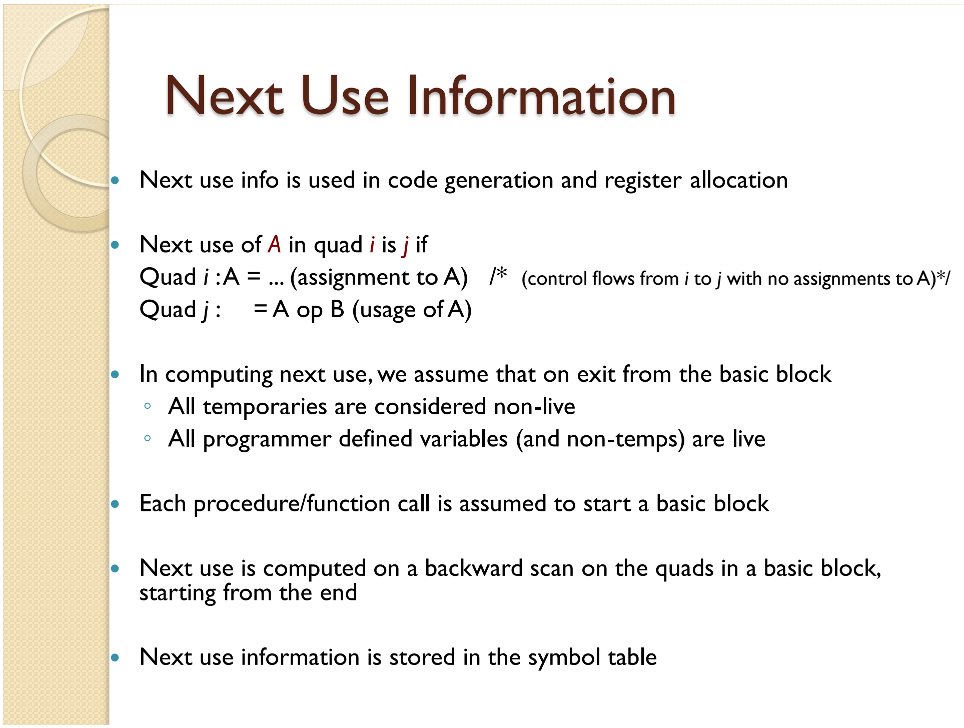
- Store only at the end of a basic block or when that register is needed for another computation
  - On exit from a basic block, store only live variables which are not in their memory locations already (use address descriptors to determine the latter)
  - If liveness information is not known, assume that all variables are live at all times



Example

- A := B+C
  - If B and C are in registers R1 and R2, then generate
    - *ADD R2,R1* (cost = 1, result in R1)
      - legal only if B is *not live* after the statement
  - If R1 contains B, but C is in memory
    - *ADD C,R1* (cost = 2, result in R1) **or**
    - *LOAD C, R2*

      ADD R2,R1 (cost = 3, result in R1)
      - legal only if B is *not live* after the statement
      - attractive if the value of C is subsequently used (it can be taken from R2)



Next Use Information

- Next use info is used in code generation and register allocation
- Next use of *A* in quad *i* is *j* if

  Quad *i* :A = ... (assignment to A) /* (control flows from *i* to *j* with no assignments to A)*/

  Quad *j* : = A op B (usage of A)
- In computing next use, we assume that on exit from the basic block
  - All temporaries are considered non-live
  - All programmer defined variables (and non-temps) are live
- Each procedure/function call is assumed to start a basic block
- Next use is computed on a backward scan on the quads in a basic block, starting from the end
- Next use information is stored in the symbol table

Scheme B – The algorithm

- We deal with one basic block at a time
- We assume that there is no global register allocation
- For each quad *A := B op C* do the following
  - Find a location *L* to perform *B op C*
    - Usually a register returned by *GETREG*() (could be a mem loc)
  - Where is *B*?
    - *B* , found using address descriptor for *B*
    - Prefer register for *B* , if it is available in memory and register
    - Generate *Load B , L* (if *B* is not in *L*)
  - Where is *C*?
    - *C* , found using address descriptor for *C*
    - Generate *op C , L*
  - Update descriptors for *L* and *A*
  - If *B*/*C* have no next uses, update descriptors to reflect this information

Function *GETREG*( )

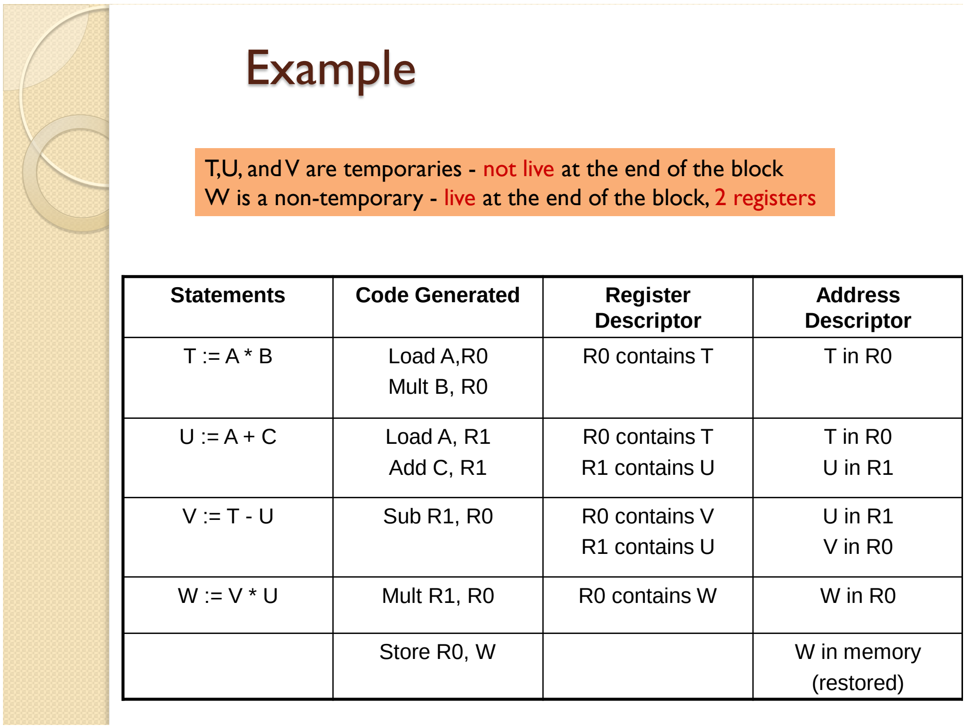
Finds *L* for computing *A := B op C*

1. If *B* is in a register (say *R*), R holds no other names, and
   - *B* has no next use, and *B* is not live after the block, then return *R*

2. Failing (1), return an empty register, if available

3. Failing (2)
   - If *A* has a next use in the block, OR
     if *B op C* needs a register (e.g., *op* is an indexing operator)
     - Use a *heuristic* to find an occupied register
       - **a register whose contents are referenced farthest in future, or**
       - **the number of next uses is smallest etc.**
     - *Spill* it by generating an instruction, *MOV R, mem*
       - ***mem* is the memory location for the variable in R that variable is not already in *mem***
     - Update *Register* and *Address* descriptors

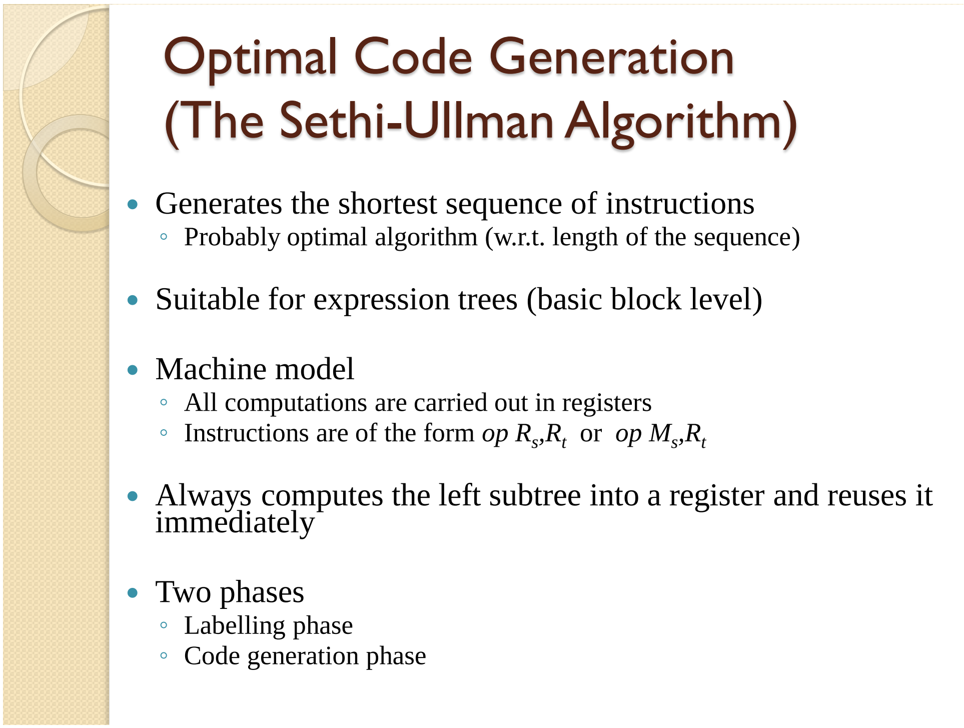
4. If *A* is not used in the block, or no suitable register can be found
   - Return a memory location for *L*



Example

T,U, and V are temporaries - not live at the end of the block
W is a non-temporary - live at the end of the block, 2 registers

Statements	Code Generated	Register Descriptor	Address Descriptor
T := A * B	Load A,R0 Mult B, R0	R0 contains T	T in R0
U := A + C	Load A, R1 Add C, R1	R0 contains T R1 contains U	T in R0 U in R1
V := T - U	Sub R1, R0	R0 contains V R1 contains U	U in R1 V in R0
W := V * U	Mult R1, R0	R0 contains W	W in R0
	Store R0, W		W in memory (restored)



Optimal Code Generation (The Sethi-Ullman Algorithm)

- Generates the shortest sequence of instructions
  - Probably optimal algorithm (w.r.t. length of the sequence)
- Suitable for expression trees (basic block level)
- Machine model
  - All computations are carried out in registers
  - Instructions are of the form *op* R_s,R_t or *op* M_s,R_t
- Always computes the left subtree into a register and reuses it immediately
- Two phases
  - Labelling phase
  - Code generation phase

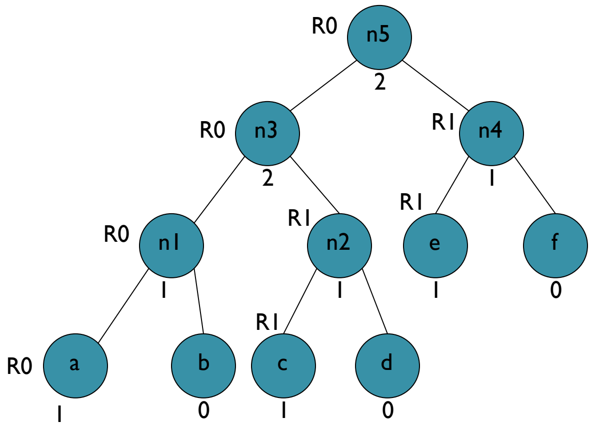
The Labelling Algorithm

- Labels each node of the tree with an integer:
  - fewest no. of registers required to evaluate the tree with no intermediate stores to memory
  - Consider binary trees
- For leaf nodes
  - ***if* n** is the leftmost child of its parent ***then***

    label(n) := 1 *else* label(n) := 0
- For internal nodes
  - **label(n) = max (l_1, l_2), if $l_1 \neq l_2$**

    = $l_1 + 1$, if $l_1 = l_2$

Labelling - Example

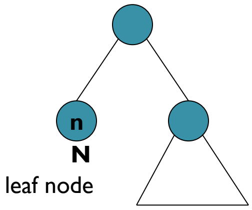


Code Generation Phase
(Procedure GENCODE(n))

- RSTACK – stack of registers, R_0,…,$R_{(r-1)}$
- TSTACK – stack of temporaries, T_0, T_1,…
- A call to Gencode(n) generates code to evaluate a tree T, rooted at node n, into the register top(RSTACK) ,and
  - the rest of RSTACK remains in the same state as the one before the call
- A swap of the top two registers of RSTACK is needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child.

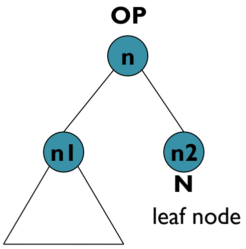
The Code Generation Algorithm (1)

```
Procedure gencode(n);
{ /* case 0 */
  if
     n is a leaf representing
     operand N and is the
     leftmost child of its parent
  then
     print(LOAD N, top(RSTACK))
```



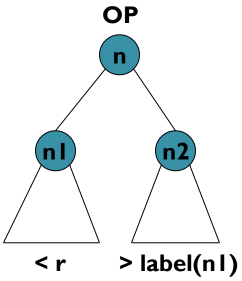
The Code Generation Algorithm (2)

```
/* case 1 */
else if
  n is an interior node with operator
  OP, left child n1, and right child n2
then
  if label(n2) == 0 then {
      let N be the operand for n2;
      gencode(n1);
      print(OP N, top(RSTACK));
   }
```



The Code Generation Algorithm (3)

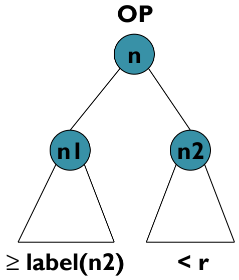
```
/* case 2 */
else if ((1 ≤ label(n1) < label(n2))
           and( label(n1) < r))
then {
   swap(RSTACK); gencode(n2);
   R := pop(RSTACK); gencode(n1);
   /* R holds the result of n2 */
   print(OP R, top(RSTACK));
   push (RSTACK,R);
   swap(RSTACK);
   }
```



The swap() function ensures that a node is evaluated into the same register as its left child

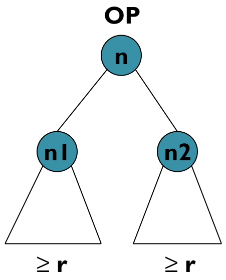
The Code Generation Algorithm (4)

```
/* case 3 */
else if (($1 \le label(n2) \le label(n1)$)
            and( $label(n2) < r$))
then {
   gencode(n1);
   R := pop(RSTACK); gencode(n2);
   /* R holds the result of n1 */
   print(OP  top(RSTACK), R);
   push (RSTACK,R);
   }
```



The Code Generation Algorithm (5)

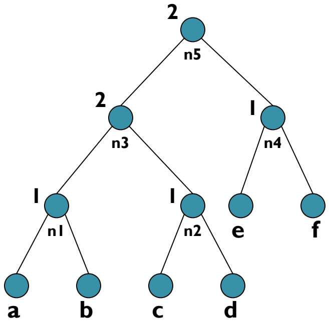
```
/* case 4, both labels are $\ge r$ */
else {
   gencode(n2);T:= pop(TSTACK);
   print(LOAD top(RSTACK),T);
   gencode(n1);
   print(OP T, top(RSTACK));
   push(TSTACK,T);
   }
}
```



Code Generation Phase – Example 1

No. of registers = r = 2

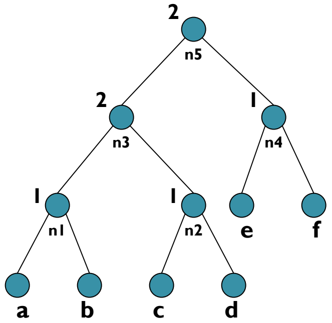
```
n5 → n3 → n1 → a → Load a, R0
                 → $op_{n1}$ b, R0
          → n2 → c → Load c, R1
                 → $op_{n2}$ d, R1
          → $op_{n3}$ R1, R0
     → n4 → e → Load e, R1
          → $op_{n4}$ f, R1
     → $op_{n5}$ R1, R0
```



Code Generation Phase – Example 2

No. of registers = r = 1.
Here we choose *rst* first so that *lst* can be computed into R0 later (case 4)

```
n5 → n4 → e → Load e, R0
          → $op_{n4}$ f, R0
     → Load R0, T0 {release R0}
     → n3 → n2 → c → Load c, R0
                 → $op_{n2}$ d, R0
          → Load R0, T1 {release R0}
          → n1 → a → Load a, R0
                 → $op_{n1}$ b, R0
          → $op_{n3}$ T1, R0 {release T1}
     → $op_{n5}$ T0, R0 {release T0}
```


2
n5
2
n3
1
n4
1
n1
1
n2
e
f
a
b
c
d