TM Simulator

CIS*4650 Compilers

(Winter 2025)

Incremental Steps for C3

- **Subtask 1**: Get familiar with TM Simulator architecture and its assembly code and test the TMSimulator package.

Basic Architecture

```
#define IADDR_SIZE 1024
#define DADDR_SIZE  1024
#define NO_REGS 8
#define PC_REG 7

INSTRUCTION iMem[IADDR_SIZE];
int dMem[DADDR_SIZE];
int reg[NO_REGS];


do {
    /* fetch an instruction */
    currentInstruction = iMem[reg[PC_REG]++];
    /* execute current instruction */
    ...
} while( !(halt || error) );
```

Runtime Errors:

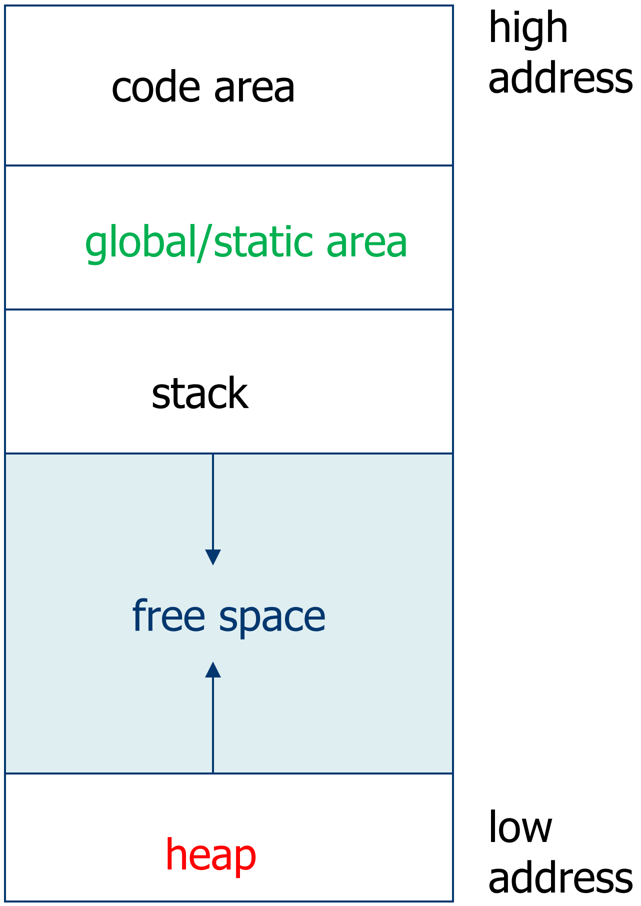
- IMEM_ERR:
  reg[PC_REG] < 0
  reg[PC_REG] >= IADDR_SIZE

- DMEM_ERR: similar to iMem above

- ZERO_DIV:

Memory Organization

- Entries to procedures and addresses to global data can be computed at compile time
- Data are often allocated at the execution time in the form of stack and/or heap
- Stack and heap can compete for the same free space or be given with separate spaces



Register Only (RO) Instructions

Format: opcode r, s, t

Opcode	*Effect*
HALT	stop execution
IN	reg[r] <- read an integer from input
OUT	reg[r] -> write to standard output
ADD	reg[r] = reg[s] + reg[t]
SUB	reg[r] = reg[s] - reg[t]
MUL	reg[r] = reg[s] * reg[t]
DIV	reg[r] = reg[s] / reg[t] (may generate ZERO_DIV)

Register Memory (RM) Instructions

Format: opcode r, d(s)

(a = d + reg[s]; may generate DMEM_ERR if a < 0 or a >= DADDR_SIZE)

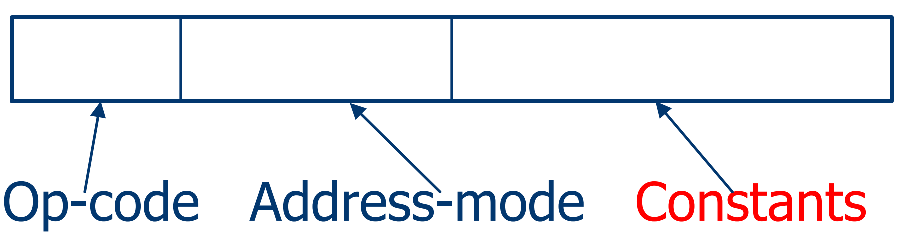
Opcode	*Effect*
LD	reg[r] = dMem[a]
LDA	reg[r] = a
LDC	reg[r] = d
ST	dMem[a] = reg[r]
JLT	if(reg[r] < 0) reg[PC_REG] = a
JLE	if(reg[r] <= 0) reg[PC_REG] = a
JGT	if(reg[r] > 0) reg[PC_REG] = a
JGE	if(reg[r] >= 0) reg[PC_REG] = a
JEQ	if(reg[r] == 0) reg[PC_REG] = a
JNE	if(reg[r] != 0) reg[PC_REG] = a

Stack-Based Runtime Environments

- Example with indirect recursion and a static variable:

```
int x = 2;

void g( int );   /* prototype */

void f( int n ) {
   static int x = 1;
   g( n );
   x--;
}
```

```
void g( int m ) {
   int y = m - 1;
   if( y > 0 ) {
      f( y );
      x--;
      g( y );
   }
}

int main( ) {
   g( x );
   return 0;
}
```



Register Memory (RM) Instructions

Format: opcode r, d(s).
(a = d + reg[s]; may generate DMEM_ERR if a < 0 or a >= DADDR_SIZE)

Opcode	*Effect*
LD	reg[r] = dMem[a]
LDA	reg[r] = a
LDC	reg[r] = d
ST	dMem[a] = reg[r]
JLT	if(reg[r] < 0) reg[PC_REG] = a
JLE	if(reg[r] <= 0) reg[PC_REG] = a
JGT	if(reg[r] > 0) reg[PC_REG] = a
JGE	if(reg[r] >= 0) reg[PC_REG] = a
JEQ	if(reg[r] == 0) reg[PC_REG] = a
JNE	if(reg[r] != 0) reg[PC_REG] = a

reg[5] = 1000
d = 22
a = d + reg[5] = 1022

1022	50

LDC 0, 22(5)
reg[0] = 22

LDA 0, 22(5)
reg[0] = 1022

LD 0, 22(5)
reg[0] = 50

Unconditional jump: goto a => LDA 7, d(s)

Sample Program

```
* This program inputs an integer, computes its factorial if it is positive,
* and prints the result

0:      IN   0, 0, 0          r0 = read
1:      JLE  0, 6(7)          if r0 > 0 then
2:      LDC 1, 1, 0             r1 = 1
3:      LDC 2, 1, 0             r2 = 1
                              * repeat
4:      MUL 1, 1, 0               r1 = r1 * r0
5:      SUB 0, 0, 2               r0 = r0 - r2
6:      JNE 0, -3(7)            until r0 == 0
7:      OUT 1, 0, 0             write r1
8:      HALT 0, 0, 0          halt

* end of program
```

Sample Program

```
* This program inputs an integer, computes its factorial if it is positive,
* and prints the result

0:      IN   0, 0, 0            r0 = read
1:      JLE  0, 6(7)            if r0 > 0 then
2:      LDC 1, 1, 0               r1 = 1
3:      LDC 2, 1, 0               r2 = 1
                                * repeat
4:      MUL 1, 1, 0                 r1 = r1 * r0
5:      SUB 0, 0, 2                 r0 = r0 - r2
6:      JNE 0, -3(7)              until r0 == 0
7:      OUT 1, 0, 0               write r1
8:      HALT 0, 0, 0            halt

* end of program
```

Three-Address vs Assembly Code

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

```
0:    IN   0, 0, 0
1:    JLE  0, 6(7)
2:    LDC 1, 1, 0
3:    LDC 2, 1, 0
4:    MUL 1, 1, 0
5:    SUB 0, 0, 2
6:    JNE 0, -3(7)
7:    OUT 1, 0, 0
8:    HALT 0, 0, 0
```

(Compared with 70 instructions "fac.tm")

Flexible Order for Backpatching

0:	IN 0, 0, 0	
2:	LDC 1, 1, 0	skip an instruction
3:	LDC 2, 1, 0	
4:	MUL 1, 1, 0	
5:	SUB 0, 0, 2	
6:	JNE 0, -3(7)	
7:	OUT 1, 0, 0	
1:	JLE 0, 6(7)	backpatching for a forward jump
8:	HALT 0, 0, 0	

Incremental Steps for C3

- **Subtask 1**: Get familiar with TM Simulator architecture and its assembly code and test the TM Simulator package.
- **Subtask 2**: Refactor the syntax trees and the visitor interface.

Required Attributes for Syntax Trees

- Declarations:

  ArrayDec: size (Already done in Checkpoint One)

  FunctionDec: funaddr (To be added)

  VarDec: super-class for SimpleDec and ArrayDec
    offset: location relative to fp or gp (To be added)
    nestLevel: gp or the current fp (To be added)

- References:

  VarExp: dtype to a VarDec (Already done in Checkpoint Two)

  CallExp: dtype to a FunctionDec (Already done in Checkpoint Two)

Extending AST's for Code Generation

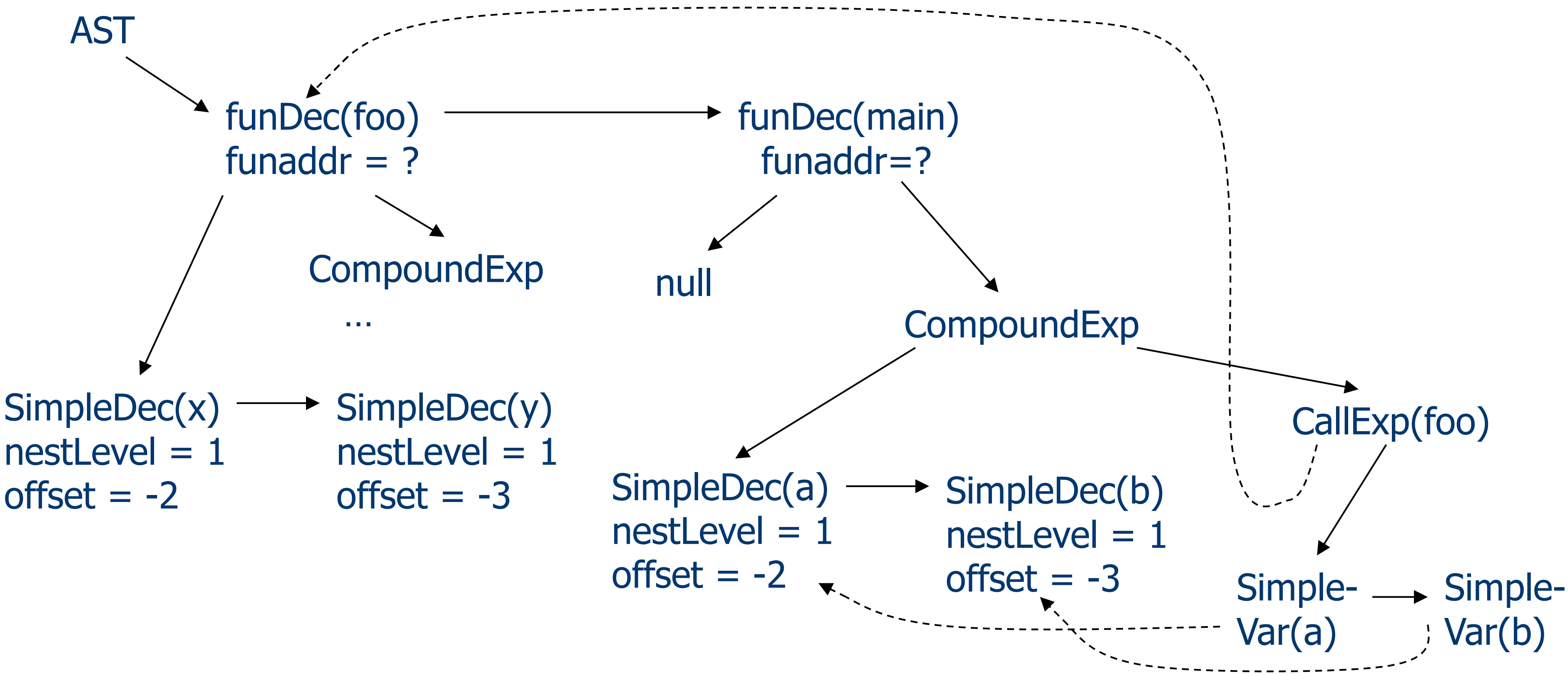
```
void foo( int x, int y ) {
   …
}
```

ofp
ret
x
y

```
void main( void ) {
   int a; int b;
   foo( a, b );
}
```

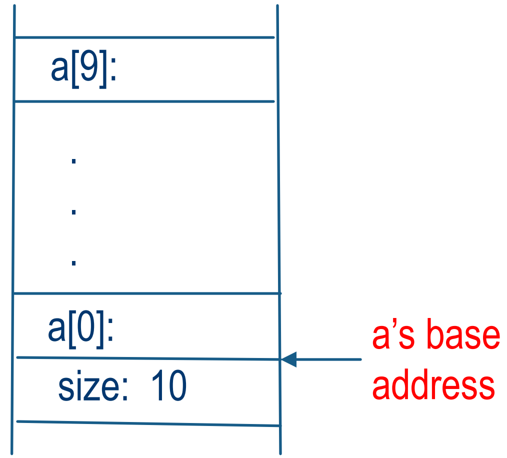
ofp	0
ret	-1
a	-2
b	-3

(ofp: also called control link)



Implementation Details

- Since an array is a static linear structure with a fixed size, we can store the "size" value right below the base address of the array
- When passing an array as an argument, the corresponding parameter will be given the base address of the array in the caller's stack frame, and right below the base address, we can access the "size" value



- Every time we access an indexed variable, we should always check the index value: if it is less than 0, we have a runtime error of "out of range below", and if it is greater or equal to "size", we have a runtime error of "out of arrange above"
- Since C- language only has integer values, we can show very large negative values such as "-1000000" for "out of range below" and "-2000000" for "out of range above" errors.

Implementation Details

For a FunctionDec, add "int funaddr" to record the start address of the corresponding function, which is needed for a function call.

For a VarDec (either SimpleDec or ArrayDec), we need to add "int nestLevel" and "int offset". The former is either 0 for "global" scope or "1" for "local" scope, and the latter is the offset within the related stackframe for memory access.

If "nestLevel = 0" and "offset = -3", we will go the global frame pointed by "gp" and its 4th location to read/write data. If "nestLevel = 1" and "offset=-2", we will go the current stackframe pointed by "fp" and access its third location (right after "ofp"and "return addr").

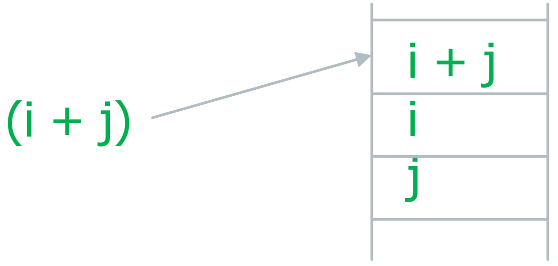
For a Var (either SimpleVar or IndexVar) and a CallExp, we need to add a link to its related definitions: SimpleDec, ArrayDec, and FunctionDec. That's where we can find the memory location or the function address.

Visitor Class for Code Generation

```
public class CodeGenerator implements AbsynVisitor {
    int mainEntry, globalOffset;
    // declare: ac, ac1, fp, gp, pc, ofpFO, retFO, initFO
    // add constructor and all emitting routines
    …
    public void visit(Absyn trees) {   // wrapper for post-order traversal
        // generate the prelude
        …
        // generate the i/o routines
        …
        // make a request to the visit method for DecList
        trees.accept(this, 0, false);
        // generate finale
        …
    }
    // implement all visit methods in AbsynVisitor such as the following
    public void visit(DecList decs, int offset, boolean isAddress) { …}
    …
}
```

Implementation Details

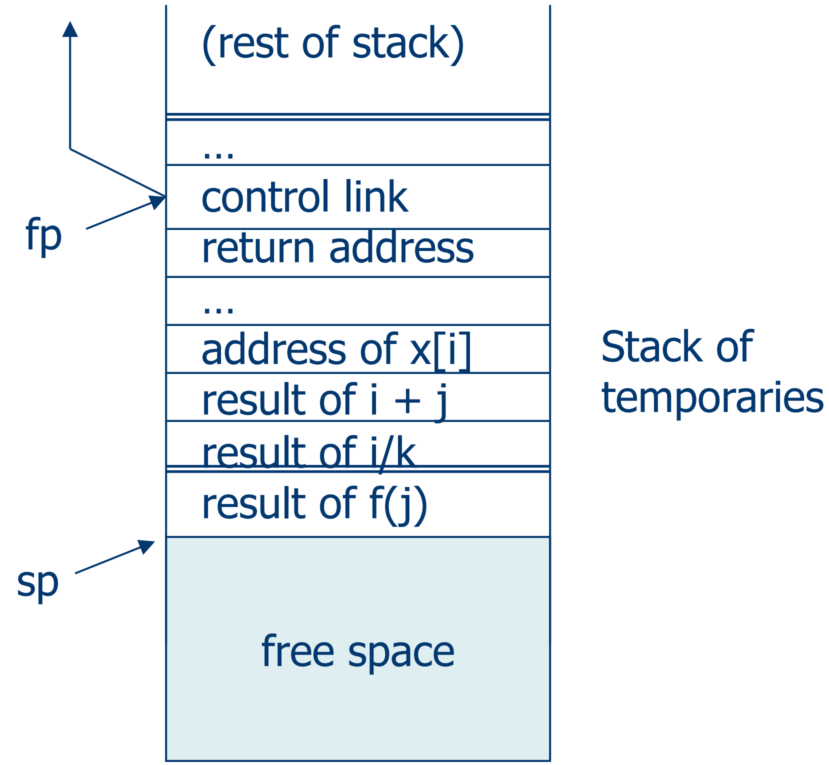
- For variable accesses, we need to handle the “visit” method differently for SimpleVar, depending on whether we are computing the left-hand side of AssignExp or not. This is distinguished by “isAddr” parameter in the “visit(Absyn tree, int offset, boolean isAddr)”.
- The value for “isAddr” is false for most cases except when calling “visit(tree.lhs, offset, true)” of AssignExp, since this is when we need to compute and save the address of a variable into a memory location.
- For the case of IndexVar, we naturally compute the address of an indexed variable, and that value can be saved directly into a memory location when used in the left-hand side of AssignExp.
- As a general principle, we use the given location to save the result of an OpExp, and the next two locations for its left and right children. In addition, register “0” is used heavily for the result, which needs to be saved to a memory location as soon as possible.



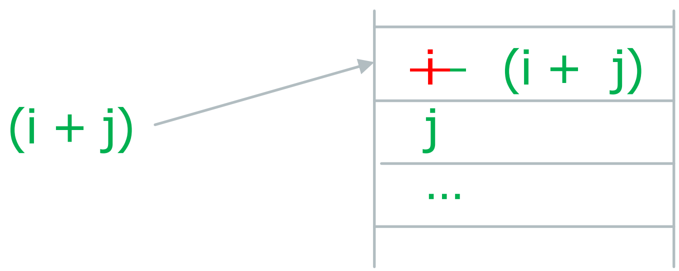
Local Temporaries

- **Computing partial results:**

$$x[i] = (i + j) + (i/k + f(j))$$



- Alternatively, we can save the result in the location of the first operand to save space:



Refactor the Visitor Pattern

- Add “boolean isAddr” to all function headers for the visitor pattern:

In Absyn.java:

```
abstract public void accept(AbsynVisitor visitor, int value, boolean flag);
```

In AbsynVisitor.java:

```
public void visit(AssignExp exp, int value, boolean flag);
…
```

In ShowTreeVisitor.java and SemanticAnalyzer.java:

```
public void visit(AssignExp exp, int level, boolean flag);
…
```

In CM.java:

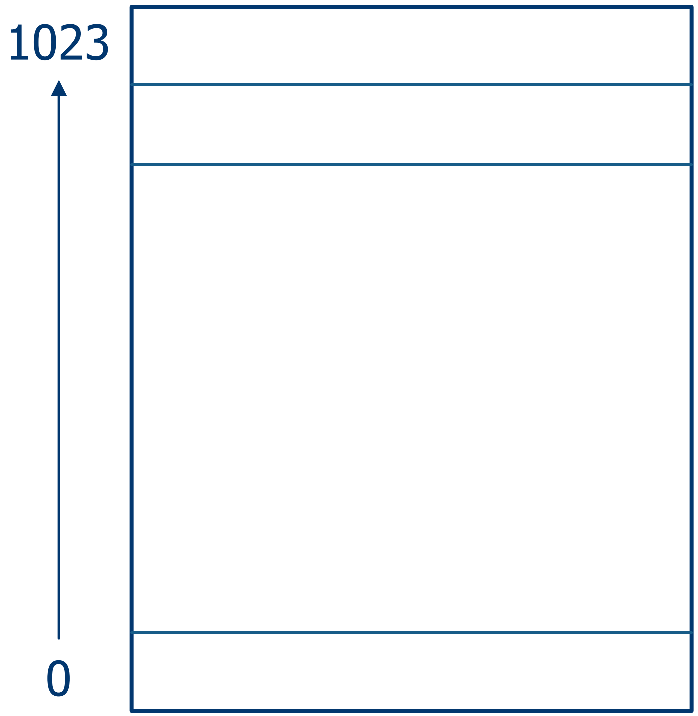
```
ShowTreeVisitor visitor = new ShowTreeVisitor();
result.accept(visitor, 0, false);
…
visitor = new CodeGenerator();
visitor.visit(result);
```

Incremental Steps for C3

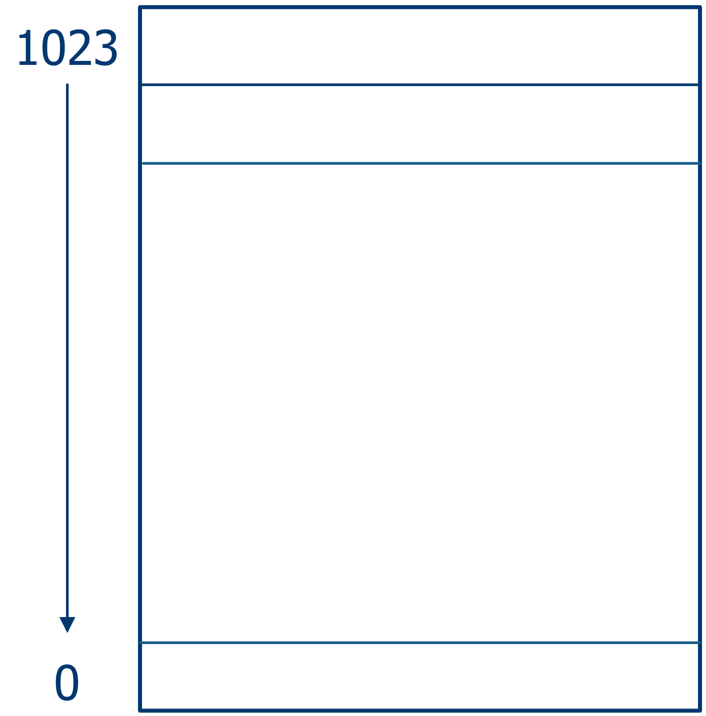
- **Subtask 1**: Get familiar with TM Simulator architecture and its assembly code and test the TM Simulator package.
- **Subtask 2**: Refactor the syntax trees and the visitor interface.
- **Subtask 3**: Implement the basic code structure that contains the code for “main” with simple declarations, expressions, and assignments only (see Slides 6-7 in the notes on “10-Intermediate Code Generation”).

Runtime Environments for C-Minus

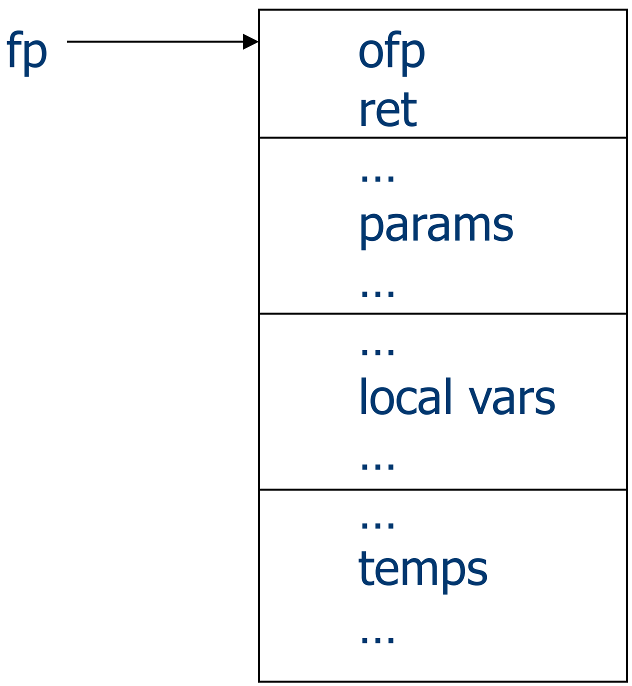
INSTRUCTION iMem[IADDR_SIZE];



int dMem[DADDR_SIZE];



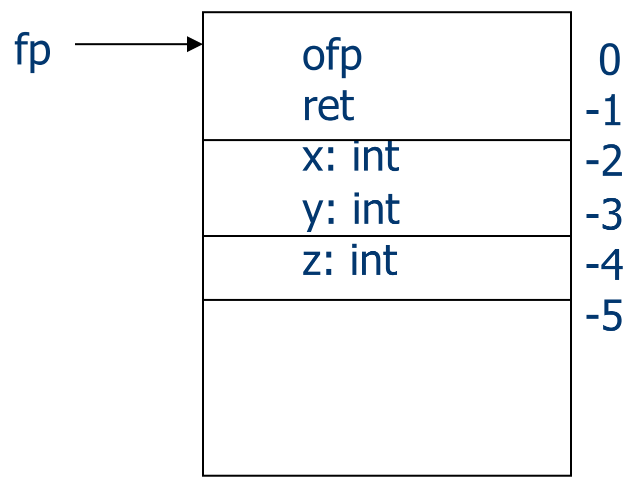
Runtime Environment for C-Minus



ofpFO = 0
retFO = -1
initFO = -2

```
int f( int x, int y ) {
   int z;
   ...
}
```

frameOffset = -5



(Note that “frameOffset” is used for “sp”, which can be passed as a parameter in the recursive functions for code generation)

Registers and Relative Addresses

- Prelude for code generation:

```
0:  LD  gp,   0(ac)       * load gp with maxaddress
1:  LDA fp,  0(gp)        * copy gp to fp
2:  ST ac,    0(ac)       * clear value at location 0
```

- Jump to a different instruction:
  - e.g., from location 42 to location 27:

```
42:  LDA  pc,   -16(pc)
```

a = (42 + 1) – 16 = 27
offset =27 – (42 + 1) = -16

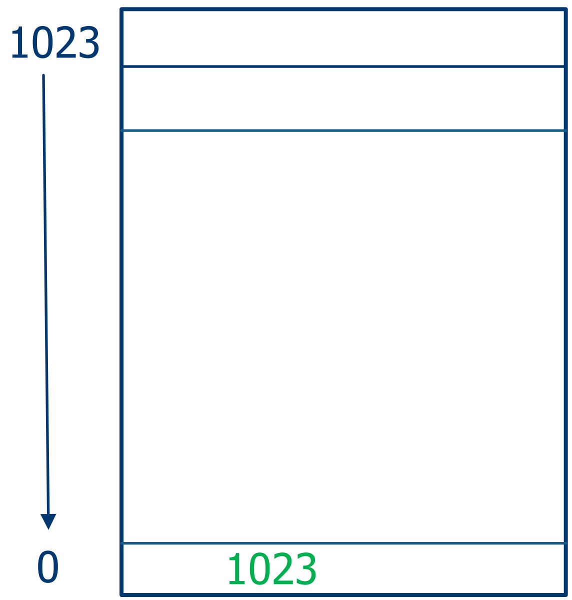
Special registers:

```
#define pc    7
#define gp    6
#define fp    5
#define ac    0
#define ac1   1
```

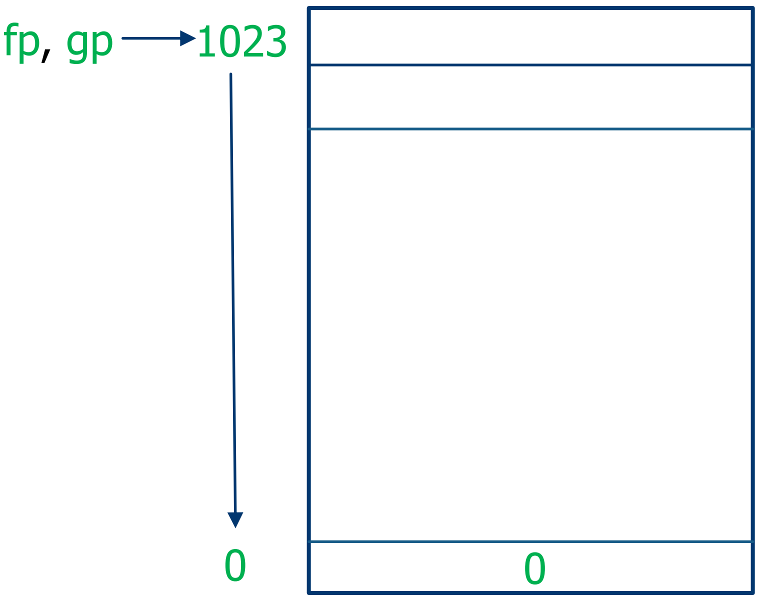
Code for Prelude

- Accessing global data:

```
0:  LD  gp,   0(ac)        * load gp with maxaddress: reg[gp] = 1023
1:  LDA fp,  0(gp)         * copy gp to fp: reg[fp] = 1023
2:  ST ac,    0(ac)        * clear location 0: dMem[0] = 0
```



(Initialization of dMem by TM Simulator)



(Initialization of dMem by TM Simulator)

Example Code

```
int y;

int gcd( int u, int v ) {
   if( v == 0 )
       return u;
   else
       return gcd( v, u – u / v * v );      // u % v = u – u/v * v
}

void main( void ) {
   int x;
   x = input();
   y = 10;
   output( gcd(x, y) );
}
```

Basic Code Structure

Lines:	Tasks to do
0-2:	prelude
3:	jump around i/o functions
4-10:	code for i/o functions
11:	jump around gcd
12-48:	code for gcd
49:	jump around main
50-80:	code for main
81-86:	finale

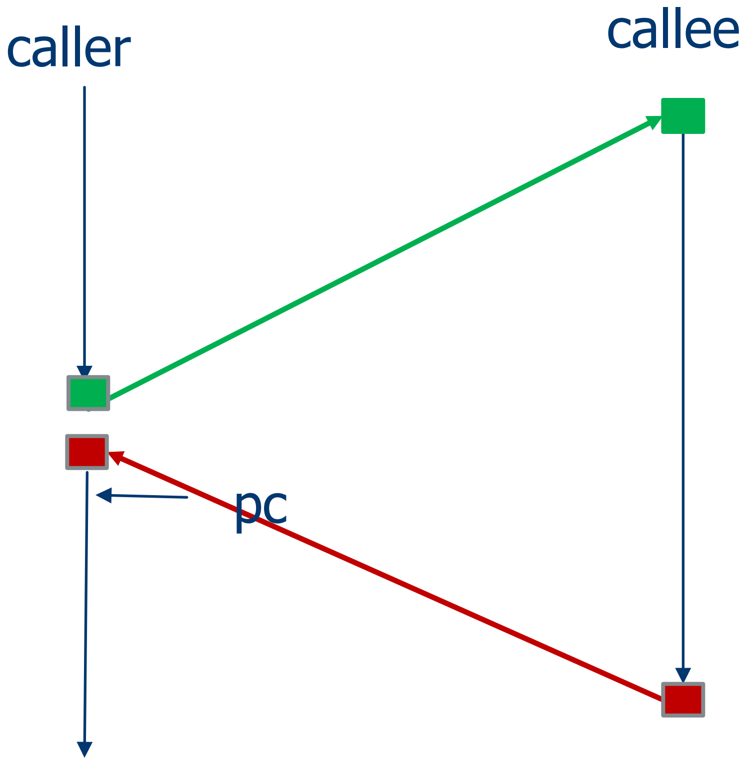
- prelude:

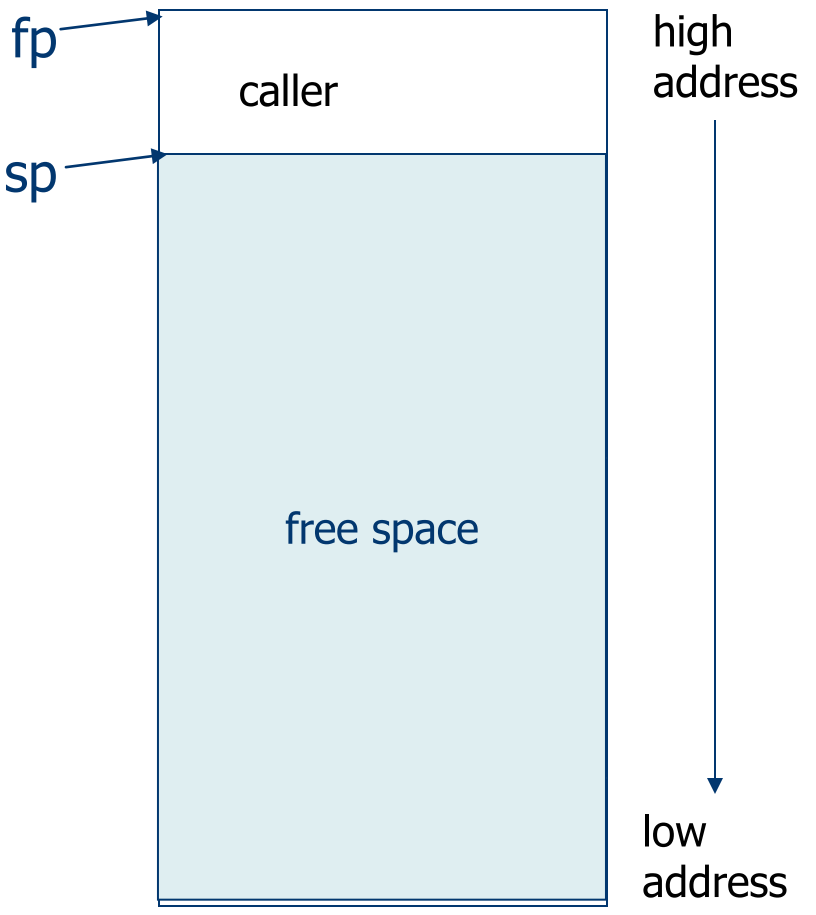
```
0:  LD  6, 0(0)    load gp with maxaddr
1:  LDA 5, 0(6)    copy gp to fp
2:  ST 0, 0(0)     clear content at loc 0
```

- finale:

```
81:  ST 5, -1(5)      push ofp
82:  LDA 5, -1(5)     push frame
83:  LDA 0, 1(7)      load ac with ret ptr
84:  LDA 7, -35(7)    jump to main loc
85:  LD 5, 0(5)       pop frame
86:  HALT 0, 0, 0
```

Calling Sequence

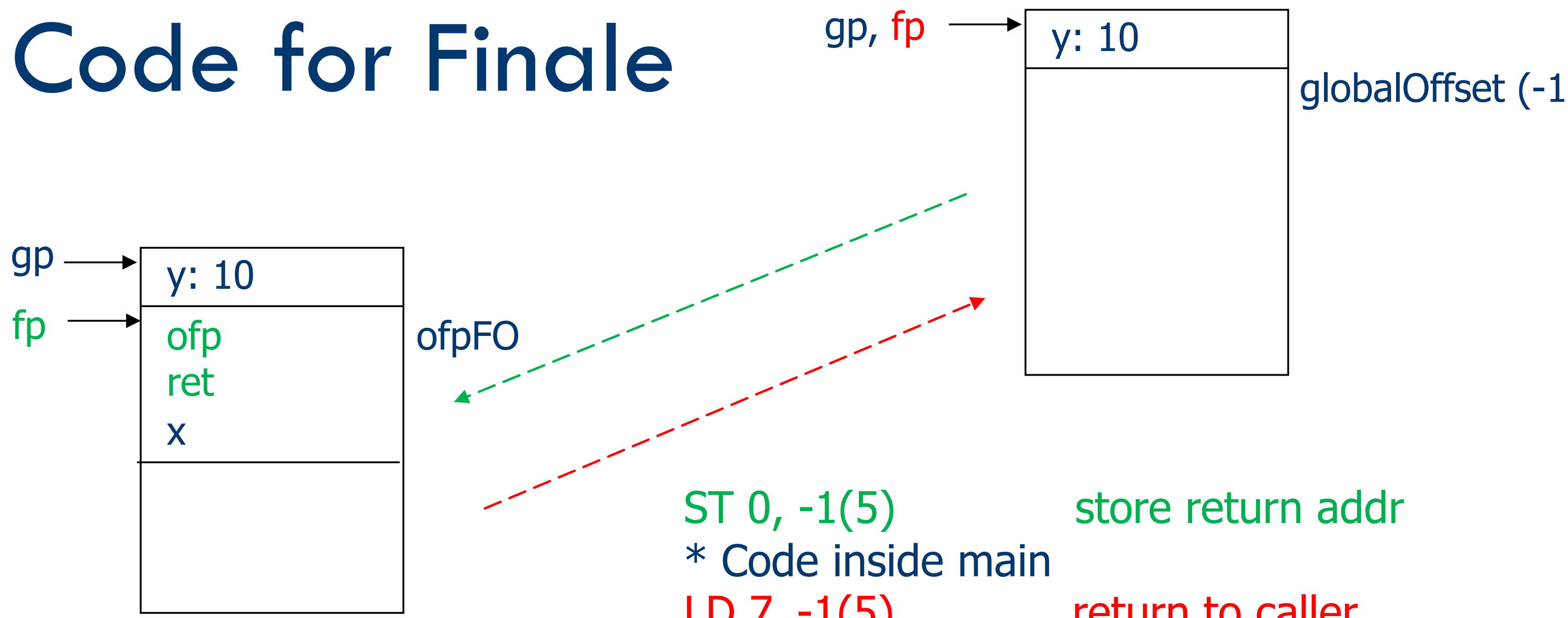




Call sequence: from caller to callee

Return sequence: from callee to caller

Code for Finale



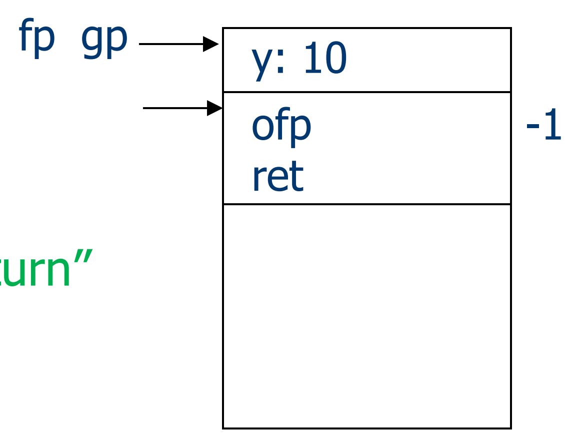
ST 0, -1(5) store return addr
* Code inside main
LD 7, -1(5) return to caller

```
ST fp, globalOffset+ofpFO (fp)
LDA fp, globalOffset (fp)
LDA ac, 1 (pc)
LDA pc, … (pc)
LD fp, ofpFO (fp)
```

```
81:  ST 5, -1(5)       push ofp
82:  LDA 5, -1(5)      push frame
83:  LDA 0, 1(7)       load ac with ret addr
84:  LDA 7, -35(7)     jump to main loc
85:  LD 5, 0(5)        pop frame
86:  HALT 0, 0, 0
```

Generating Finale

```
int mainEntry;                /* absolute address for main */
int globalOffset;              /* next available loc after global frame */

…
emitRM( “ST”, fp, globalOffset+ofpFO, fp, “push ofp” );
emitRM( “LDA”, fp, globalOffset, fp, “push frame” );
emitRM( “LDA”, ac, 1, pc, “load ac with ret ptr” );
emitRM_Abs( “LDA”, pc, mainEntry, “jump to main loc” );
emitRM( “LD”, fp, ofpFO, fp, “pop frame” );
emitRO( “HALT”, 0, 0, 0, “” );
```



- Beginning of a function: “ST 0, -1(5)   store return”

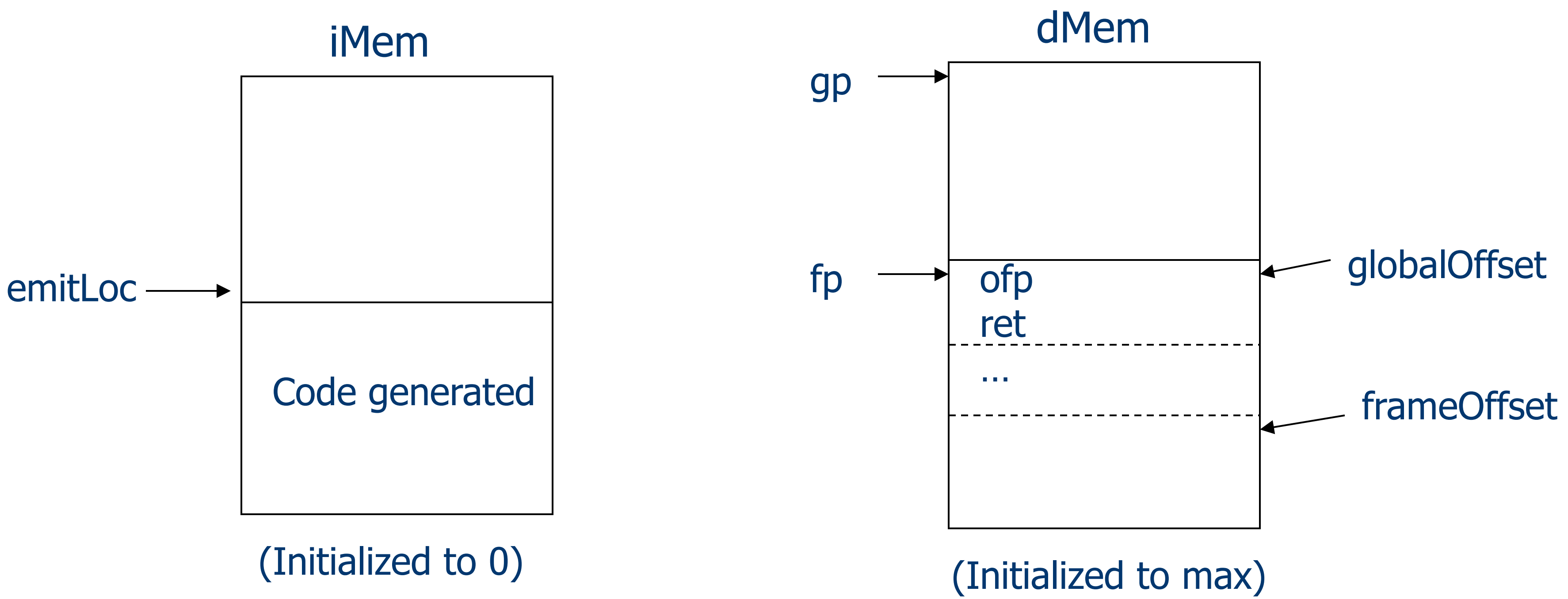
```
emitRM( “ST”, ac, retFO, fp, “store return” );
```

Code Emitting Routines

```
/* Routines to generate different kinds
   of assembly instructions
*/

int emitLoc = 0;
int highEmitLoc = 0;


void emitRO( char *op,
   int r, int s, int t, char *c ) {
   fprintf( code, "%3d: %5s %d, %d, %d",
      emitLoc, op, r, s, t );
   fprintf( code, "\t%s\n", c );
   ++emitLoc;
   if( highEmitLoc < emitLoc )
      highEmitLoc = emitLoc;
}
```

```
void emitRM( char *op,
   int r, int d, int s, char *c ) {
   fprintf( code, "%3d: %5s %d, %d(%d)",
      emitLoc, op, r, d, s );
   fprintf( code, "\t%s\n", c );
   ++emitLoc;
   if( highEmitLoc < emitLoc )
      highEmitLoc = emitLoc;
}

void emitRM_Abs( char *op,
   int r, int a, char *c ) {
   fprintf( code, "%3d:  %5s %d, %d(%d) ",
      emitLoc, op, r, a – (emitLoc + 1), pc );
   fprintf( code, "\t%s\n", c );
   ++emitLoc;
   if( highEmitLoc < emitLoc )
      highEmitLoc = emitLoc;
}
```

Three Offsets to Keep Track of



- Main function is the last declaration in a program: that's where we set values for "mainEntry" and "globalOffset".

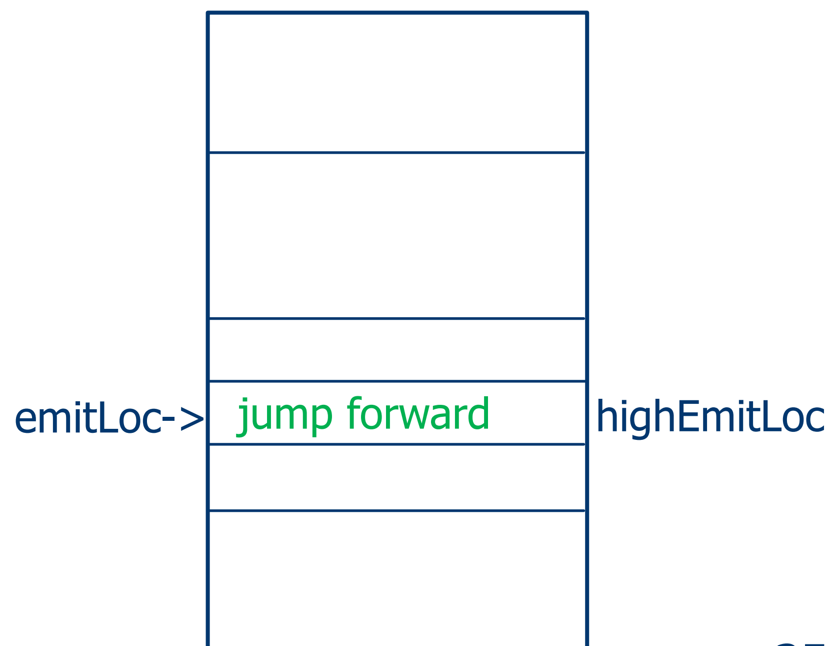
Code Emitting Routines

```
/* Routines to maintain the code space */

int emitSkip( int distance ) {
   int i = emitLoc;
   emitLoc += distance;
   if( highEmitLoc < emitLoc )
       highEmitLoc = emitLoc;
   return i;
}

void emitBackup( int loc ) {
   if( loc > highEmitLoc )
       emitComment( “BUG in emitBackup” );
   emitLoc = loc;
}

void emitRestore( void ) {
   emitLoc = highEmitLoc;
}
```

```
/* Routine to generate one line
   of comment
*/

void emitComment( char *c ) {
   fprintf( code, “* %s\n”, c );
}
```

Code Emitting Routines

```
/* functions to maintain code space: some methods like emitRO, emitRM, and
    emitComment need to be added
 */
static int emitLoc = 0;
static int highEmitLoc = 0;

int emitSkip( int distance ) {
   int i = emitLoc;
   emitLoc += distance;
   if( highEmitLoc < emitLoc )
       highEmitLoc = emitLoc;
   return i;
}

void emitBackup( int loc ) {
   if( loc > highEmitLoc )
       emitComment( "BUG in emitBackup" );
   emitLoc = loc;
}

void emitRestore( void ) {
   emitLoc = highEmitLoc;
}
```



Backpatching Example

```
* Standard prelude
0:  LD 6, 0(0)
1:  LDA 5, 0(6)
2:  ST 0, 0(0)
* Jump around i/o routines
* Code for input routine
4:  ST 0, -1(5)        store return
5:  IN 0, 0, 0         input
6:  LD 7, -1(5)        return to caller
* Code for output routine
7:  ST 0, -1(5)        store return
8:  LD 0, -2(5)        load output value
9:  OUT 0, 0, 0        output
10: LD 7, -1(5)        return to caller
3:  LDA 7, 7(7)        jump around i/o code
11:
```

```
/* code for backpatching */
…
int savedLoc = emitSkip(1);   // 3
/*
   code for the i/o routines
*/
int savedLoc2 = emitSkip(0);   // 11
emitBackup( savedLoc );
emitRM_Abs( "LDA", pc, savedLoc2, "" );
emitRestore();
…
```

Three-Address vs Assembly Code

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

```
0:      IN   0, 0, 0
1:      JLE  0, 6(7)
2:      LDC 1, 1, 0
3:      LDC 2, 1, 0
4:      MUL 1, 1, 0      int saved = emitSkip(0);
5:      SUB 0, 0, 2
6:      JNE 0, -3(7)     emitRM_Abs("LDA",pc,saved,"");
7:      OUT 1, 0, 0
8:      HALT 0, 0, 0
```

(Compared with 70 instructions "fac.tm")

Backpatching for forward jumps

Implementation Details

(1) For the very initial implementation, we can use an empty main function such as "void main(void) {}". This should lead to the following sequence of instructions:

```
/* code for prelude */
…
/* code for i/o routines */
…
12: ST 0, -1(5)        save return address
13: LD 7, -1(5)        return back to the caller
11: LDA 7, 2(7)        jump forward to finale
 /* code for finale */
 …
```

- After that, we can gradually add code generation functions for expressions, assignments, control structures, functions and recursions, nested blocks, arrays, and runtime error checking, as suggested in the marking scheme for Checkpoint Three.

Basic Code Structure

Lines:	Tasks to do
0-2:	prelude
3:	jump around i/o functions
4-10:	code for i/o functions
11:	jump around gcd
12-48:	code for gcd
49:	jump around main
50-80:	code for main
81-86:	finale

- prelude:

```
0:  LD  6, 0(0)    load gp with maxaddr
1:  LDA 5, 0(6)    copy gp to fp
2:  ST 0, 0(0)     clear content at loc 0
```

- finale:

```
81:  ST 5, -1(5)       push ofp
82:  LDA 5, -1(5)      push frame
83:  LDA 0, 1(7)       load ac with ret ptr
84:  LDA 7, -35(7)     jump to main loc
85:  LD 5, 0(5)        pop frame
86:  HALT 0, 0, 0
```

Implementation Details

(2) Implement all the emit routines in CodeGenerator.java:

- Note that every time we emit an instruction, “emitLoc” is always incremented, and if it exceeds “highEmitLoc”, the latter is also adjusted up.

- When generating an instruction, avoid using ”fprintf” directly, but use the related “emit” routine since we can increment “emitLoc” and “highEmitLoc” as well.

```
void emitRO( char *op,
   int r, int s, int t, char *c ) {
   fprintf( code, "%3d: %5s %d, %d, %d",
      emitLoc, op, r, s, t );
   fprintf( code, "\t%s\n", c );
   ++emitLoc;
   if( highEmitLoc < emitLoc )
       highEmitLoc = emitLoc;
}
```

Implementation Details

(3) Maintain three different offsets during code generation:

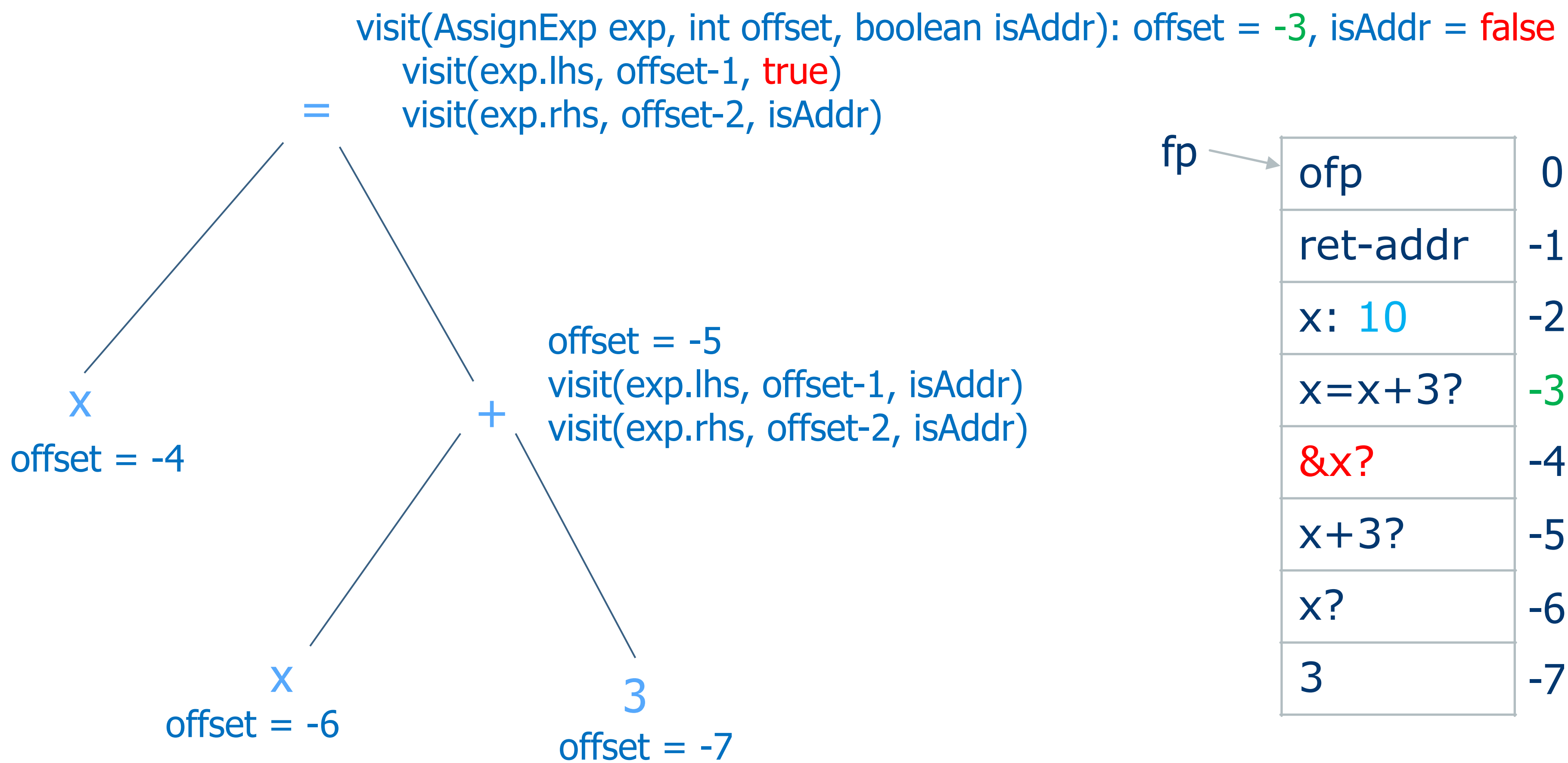
- For iMem, declare “emitLoc” and “highEmitLoc” as global variables. The former points to the current instruction we are generating (may go back to an earlier location for backpatching), while the latter always points to the next available space so that we can continue adding new instructions.

- The global stackframe at the top of dMem is pointed by the “gp” register, and its bottom is indicated by the global variable “globalOffset”. If we have ”int a” and “int b” declared in the global scope, we will have “globalOffset=-2”. If we have “int x[10]” declared in the global scope, we will have “globalOffset=-11” (10 integers plus 1 more for size).

- The current stackframe in dMem is pointed by the “fp” register, and its bottom is indicated by a parameter “frameOffset”, which is local in your recursive function for code generation. Since the first two locations are reserved for “ofp” and “return addr”, the parameters and local variables will start from “-2” location (initFO) in the stackframe.

Maintaining “frameoffset”

visit(tree, -3, false): initial call with the syntax tree for “x = x + 3;”

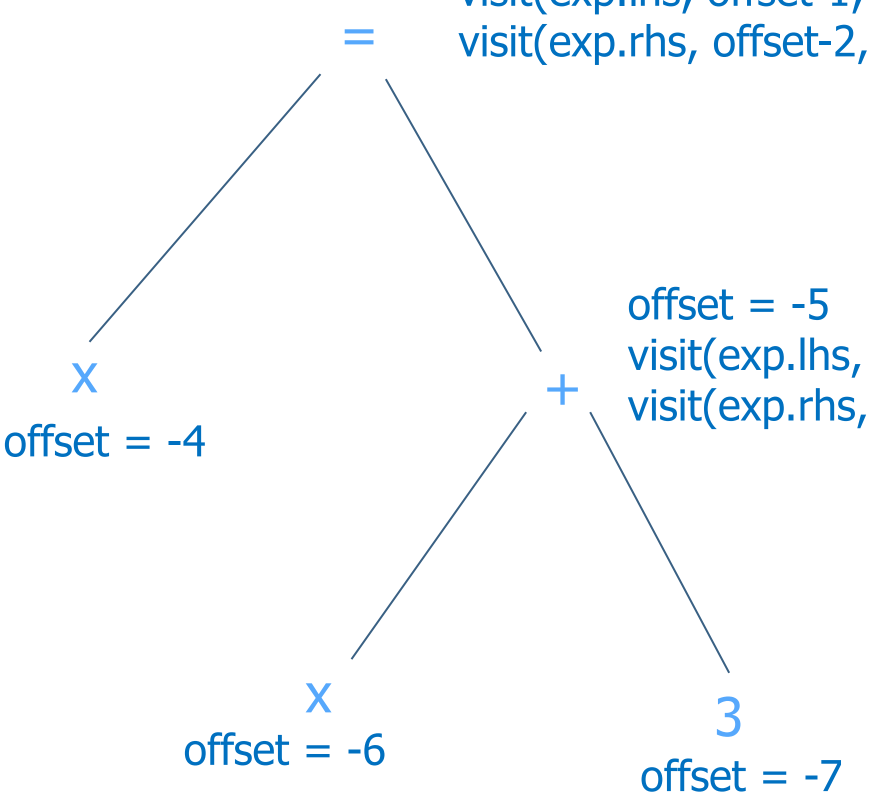


Maintaining "frameoffset"

visit(tree, -3, false): initial call with the syntax tree for "x = x + 3;"

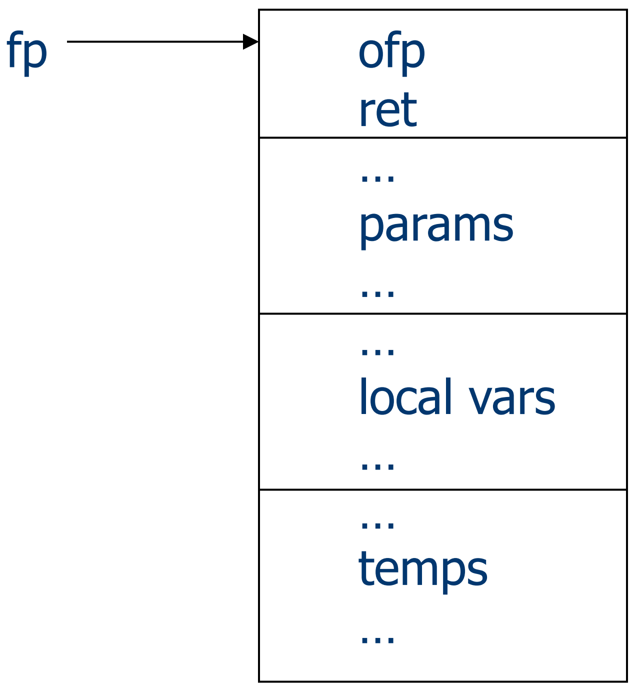
```
visit(AssignExp exp, int offset, boolean isAddr): offset = -3, isAddr = false
    visit(exp.lhs, offset-1, true)
    visit(exp.rhs, offset-2, isAddr)
```

```
offset = -5
visit(exp.lhs, offset-1, isAddr)
visit(exp.rhs, offset-2, isAddr)
```



fp	ofp	0
	ret-addr	-1
	x: ~~10~~ 13	-2
	x=x+3:13	-3
	&x: -2(5)	-4
	x+3:13	-5
	x:10	-6
	3	-7

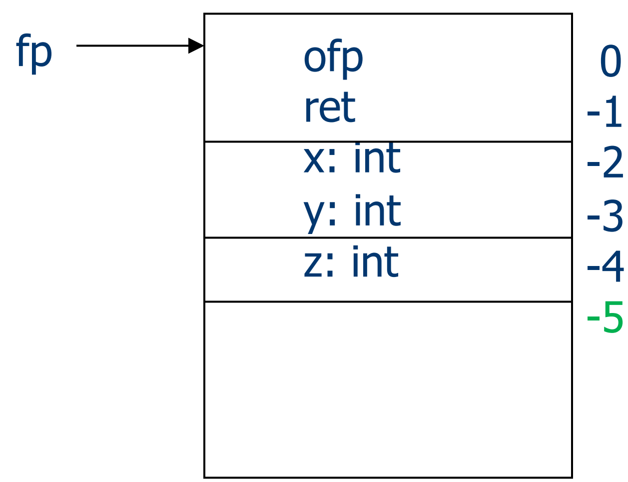
Runtime Environment for C-Minus



ofpFO = 0
retFO = -1
initFO = -2

```
int f( int x, int y ) {
   int z;
   …
}
```

frameOffset = -5



(Note that "frameOffset" is used for "sp", which can be passed as a parameter in the recursive functions for code generation)

Implementation Details

(4) Slides 6-7 in the notes on "10-CodeGeneration" show the key steps for generating intermediate code for expressions. For Checkpoint Three, however, we need to map them further to generate TM assembly code. This can be illustrated with the following example:

- Assuming the syntax tree for the expression "x = x + 3" and the stack frame on the right.
- The initial call is "visit(tree, -3, false)", where "tree" is an AssignExp and "-3" is the frameOffset.
- Inside the "visit" for AssignExp, we will call "visit(tree.lhs, -4, true)" and "visit(tree.rhs, -5, false) first in the post-order traversal. The former is for a SimpleVar and the latter is for an OpExp.
- Inside the "visit" for SimpleVar when used as the left-hand side of AssignExp, we will compute the address of "x" and save it in location "-4". This is done with these two instructions (assuming that we are starting with instruction 13):

  13: LDA 0, -2(5)   and   14: ST 0, -4(5)

fp →

ofp	0
ret-addr	-1
x: 10	-2
	-3
&x:-2(5)	-4
	-5
	-6
	-7

Implementation Details

- Inside the "visit" for OpExp, we will call "visit(tree.left, -6, false)" and "visit(tree.right, -7, false)" in the post-order traversal. The former is for a SimpleVar and the latter is for an IntExp.
- Inside the "visit" for SimpleVar when not used as the left-hand side of AssignExp, we simply save the value of "x" to location "-6" with the following two instructions:
  15: LD 0, -2(5) and 16: ST 0, -6(5)
- Inside the "visit" for IntExp, we will save the value of "3" to location "-7" with these instructions:
  17: LDC 0, 3(0) and 18: ST 0, -7(5)
- Back to the "visit" for OpExp, we will do the addition save the result in location "-5" with these instructions:
  19: LD 0, -6(5) and 20: LD 1, -7(5)
  21: ADD 0, 0, 1 and 22: ST 0, -5(5)
- Back to the "visit" for AssignExp, we will do the assignment and save the result to location "-3" with the following instructions:
  23: LD 0, -4(5) and 24: LD 1, -5(5) and 25: ST 1, 0(0)
  26: ST 1, -3(5)

fp →	ofp	0
	ret-addr	-1
	x:x+3 = 13	-2
	x + 3 = 13	-3
	&x: -2(5)	-4
	result of x+3:13	-5
	value of x:10	-6
	value of 3	-7

Incremental Steps for C3

- **Subtask 1**: Get familiar with TM Simulator architecture and its assembly code and test the TM Simulator package.
- **Subtask 2**: Refactor the syntax trees and the visitor interface.
- **Subtask 3**: Implement the basic code structure that contains the code for “main” with simple variable declarations, expressions, and assignments only (see Slides 6-7 in the notes on “10-Intermediate Code Generation”).

Incremental Steps for C3

- **Subtask 1**: Get familiar with TM Simulator architecture and its assembly code and test the TM Simulator package.
- **Subtask 2**: Refactor the syntax trees and the visitor interface.
- **Subtask 3**: Implement the basic code structure that contains the code for "main" with simple declarations, expressions, and assignments only (see Slides 6-7 in the notes on "10-Intermediate Code Generation").
- **Subtask 4**: Implement the code generation for control structures (see Slides 15-21 in the notes on "10-Intermediate Code Generation): able to test "fac.cm" and "booltest.cm".
- **Subtask 5**: Implement the calling sequence fully: able to call input/output functions and test "gcd.cm" and "mutual.cm".

Backpatching Example

```
* Standard prelude
0:  LD 6, 0(0)
1:  LDA 5, 0(6)
2:  ST 0, 0(0)
* Jump around i/o routines
* Code for input routine
4:  ST 0, -1(5)       store return
5:  IN 0, 0, 0        input
6:  LD 7, -1(5)       return to caller
* Code for output routine
7:  ST 0, -1(5)       store return
8:  LD 0, -2(5)       load output value
9:  OUT 0, 0, 0       output
10: LD 7, -1(5)       return to caller
3:  LDA 7, 7(7)       jump around i/o code
11:
```

```
/* code for backpatching */
…
int savedLoc = emitSkip(1);   //3
/*
   code for the i/o routines
*/
int savedLoc2 = emitSkip(0);   // 11
emitBackup( savedLoc );
emitRM_Abs( "LDA", pc, savedLoc2, "" );
emitRestore();
…
```

Three-Address vs Assembly Code

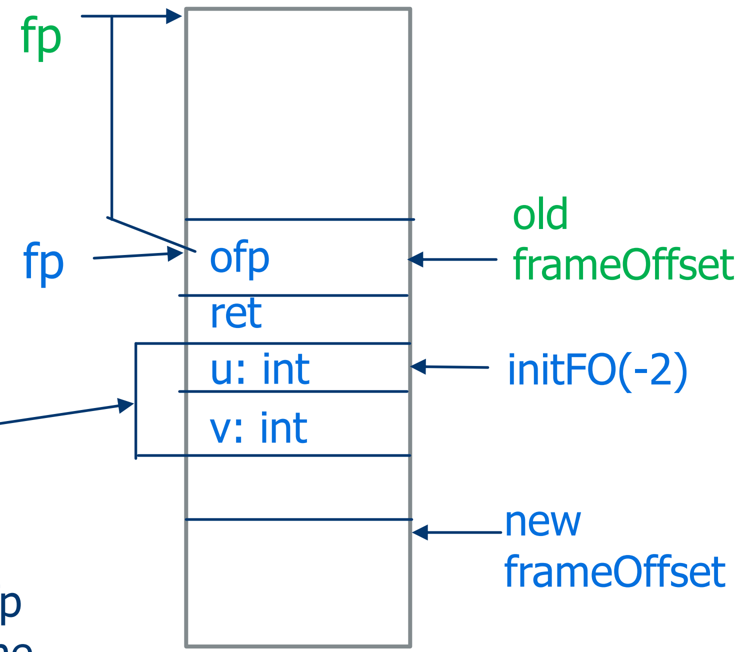
```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

```
0:    IN   0, 0, 0
1:    JLE  0, 6(7)
2:    LDC 1, 1, 0
3:    LDC 2, 1, 0
4:    MUL 1, 1, 0       int saved = emitSkip(0);
5:    SUB 0, 0, 2
6:    JNE 0, -3(7)      emitRM_Abs("LDA", pc, saved,"");
7:    OUT 1, 0, 0
8:    HALT 0, 0, 0
```

(Compared with 70 instructions "fac.tm")

Backpatching for forward jumps

Calling Sequence



- Call sequence:

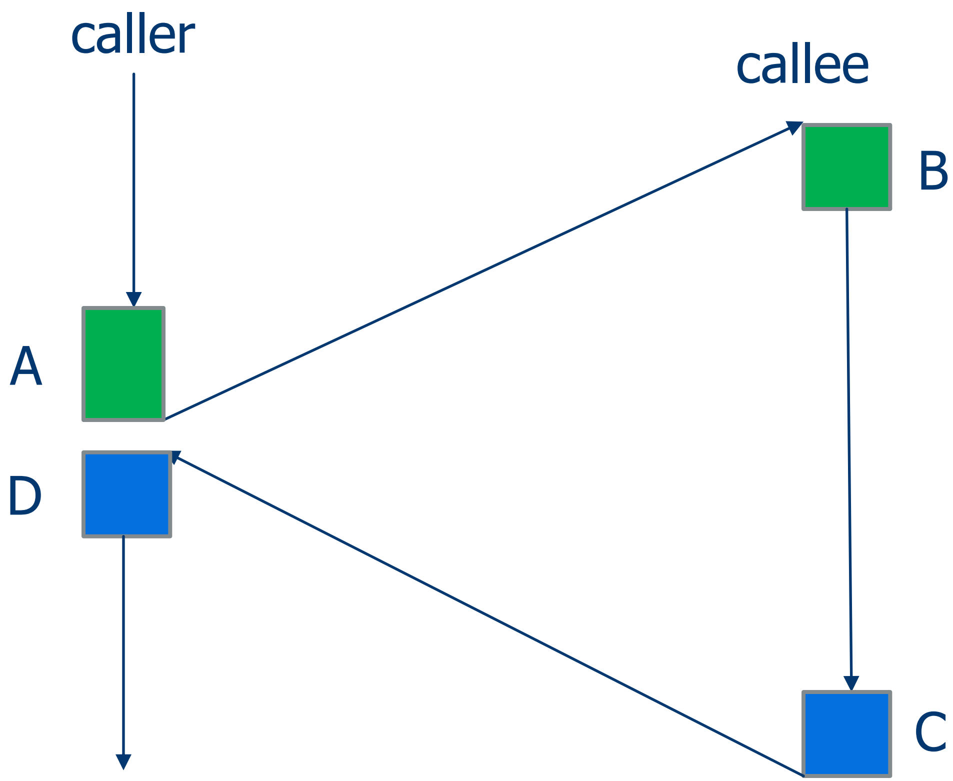
pc →

```
<code to compute first arg>
ST ac, frameOffset+initFO (fp)
<code to compute second arg>
ST ac, frameOffset+initFO-1 (fp)
ST fp, frameOffset+ofpFO (fp)      * store current fp
LDA fp, frameOffset (fp)           * push new frame
LDA ac, 1 (pc)                     * save return in ac
LDA pc, … (pc)                     * relative jump to function entry
LD fp, ofpFO (fp)                  * pop current frame
```

- Return sequence:

```
ST ac, retFO (fp)                  * store return address
…
LD pc, retFO (fp)                  * return to caller
```

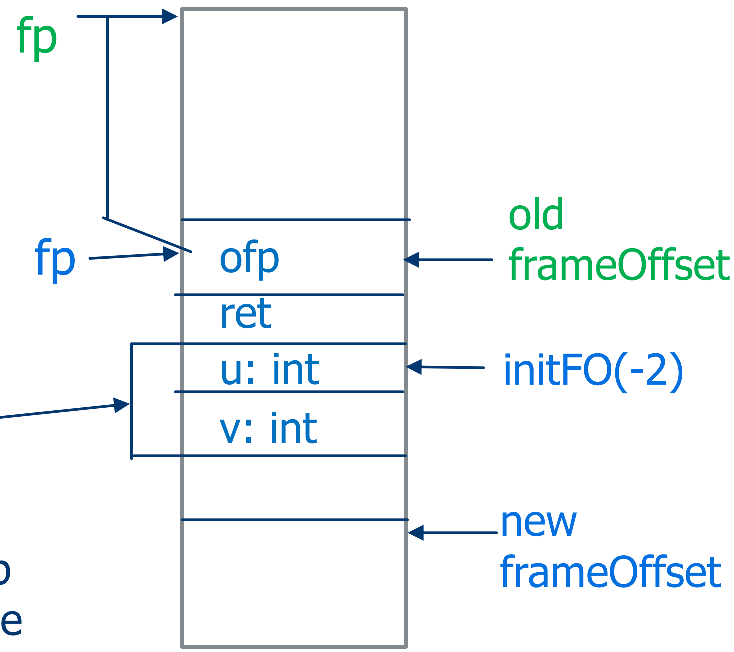
Calling Sequence



Function Returns

- **Call sequence:**

```
pc → <code to compute first arg>
     ST ac, frameOffset+initFO (fp)
     <code to compute second arg>
     ST ac, frameOffset+initFO-1 (fp)
     ST fp, frameOffset+ofpFO (fp)      * store current fp
     LDA fp, frameOffset (fp)           * push new frame
     LDA ac, 1 (pc)                     * save return in ac
     LDA pc, … (pc)                     * relative jump to function entry
     LD fp, ofpFO (fp)                  * pop current frame
     ST ac, frameOffset(fp)             * save return value to caller's stack frame
```



- **Return sequence:**

```
ST ac, retFO (fp)                   * store return address
…
// save return value to "ac"
LD pc, retFO (fp)                   * return to caller
```

```
Callee:
int foo(int x) {
  return x+1;
}

x = 3 + foo(2)
```

Calling Sequence vs Finale

- calling sequence:

```
<code to compute first arg>
ST ac, frameOffset+initFO (fp)
<code to compute second arg>
ST ac, frameOffset+initFO-1 (fp)
ST fp, frameOffset+ofpFO (fp)
LDA fp, frameOffset (fp)
LDA ac, 1 (pc)
LDA pc, … (pc)
LD fp, ofpFO (fp)
```

- finale:

```
ST fp, globalOffset+ofpFO (fp)   push ofp
LDA fp, globalOffset (fp)        push frame
LDA ac, 1(pc)                    load ac with ret
LDA pc, … (pc)                   jump to main
LD fp, ofpFO (fp)                pop frame
HALT 0, 0, 0
```

Finale vs Calling Sequence

Similarities:

Both are about the function calls and returns

Differences: for finale

- call the “main” only
- no parameter passing
- relative to globalOffset
- ended with “halt”

Differences: for regular function

- call a regular function
- may do parameter passing
- relative to frameOffset
- may need to a return value

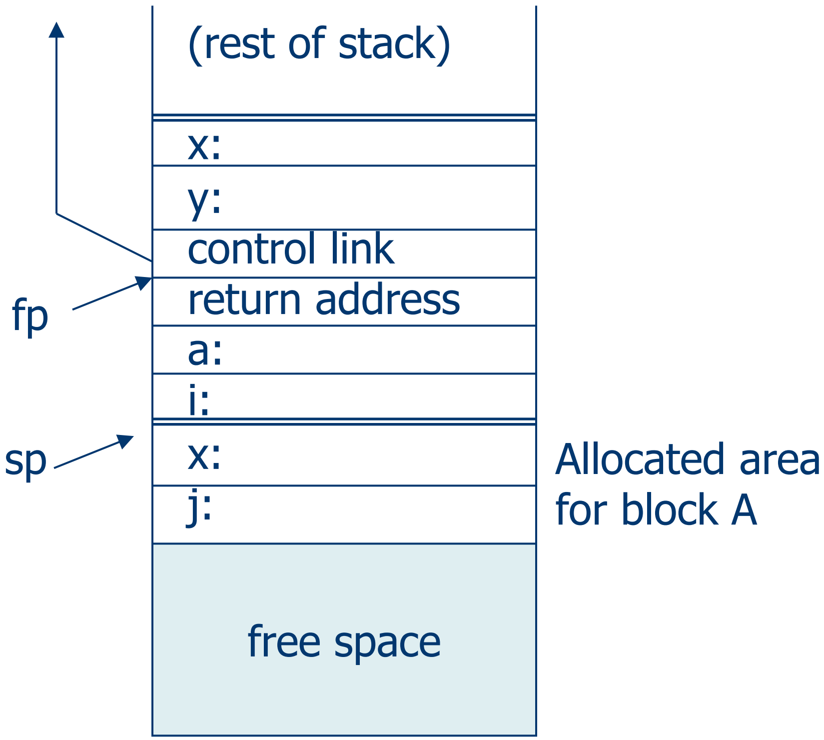
Incremental Steps for C3

- **Subtask 1**: Get familiar with TM Simulator architecture and its assembly code and test the TM Simulator package.
- **Subtask 2**: Refactor the syntax trees and the visitor interface.
- **Subtask 3**: Implement the basic code structure that contains the code for "main" with simple declarations, expressions, and assignments only (see Slides 6-7 in "10-Intermediate Code Generation").
- **Subtask 4**: Implement the code generation for control structures (see Slides 15-21 in"10-Intermediate Code Generation): able to test "fac.cm".
- **Subtask 5**: Implement the calling sequence fully: able to call input/output functions and test "gcd.cm".
- **Subtask 6**: Implement the code generation for inner blocks and array references: able to test "sort.cm" (see Slides 8-12 in "10-Intermediate Code Generation")
- **Subtask 7**: Implement runtime error checking of the array index: index out of bound either below or above the given range

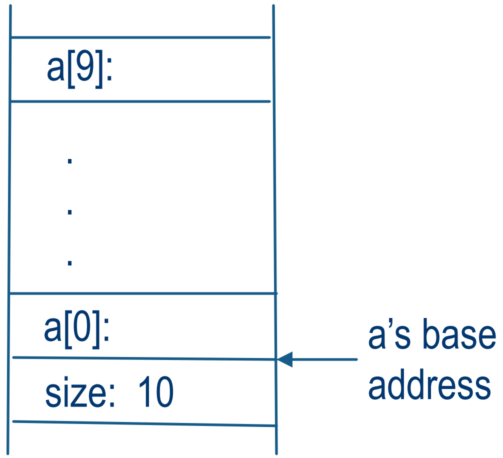
Nested Blocks

- A simple solution is to allocate temporaries on entry to a block and de-allocate them on exit

```
void p( int x, double y ) {
   char a;
   int i;
   …
   A: { double x;
          int j;
          …
       }
       …
   B: { char * a;
          int k;
          …
      }
      …
}
```



Implementation Detials



- Since an array is a static linear structure with a fixed size, we can store the “size” value right below the base address of the array
- When passing an array as an argument, the corresponding parameter will be given the base address of the array in the caller’s stack frame, and right below the base address, we can access the “size” value
- Every time we access an indexed variable, we should always check the index value: if it is less than 0, we have a runtime error of “out of range below”, and if it is greater or equal to “size”, we have a runtime error of “out of arrange above”
- Since C- language only has integer values, we can show very large negative values such as “-1000000” for “out of range below” and “-2000000” for ”out of range above” errors.

Code for Logical Expressions

- Short circuit:
  - If a is false then (a and b) is also false
  - If a is true then (a or b) is also true
- If-expressions: equivalent to if-statements except that they return values
  - a and b ≡ if a then b else false
  - a or b ≡ if a then true else b

```
(x != 0) && (y == x)

if( x != 0 ) then (y == x)
else false
```

```
t1 = (x != 0)
if_false t1 goto L1
t2 = (y == x)
goto L2
label L1
t2 = FALSE
label L2
```

Special Cases for Functions

- **Missing "main" function**: In CodeGenerator.java, we need to maintain an instance variable "mainEntry", which can be initialized to "-1". If its value remains to be "-1" when we need to generate the code for "finale", we can report an error for missing the "main" function and terminate the code generation process.

- **Accessing input/output functions:** Since these are predefined functions, we should always generate the related code right after the step "prelude". To access them later, however, we need to maintain two more instance variables "inputEntry" and "outputEntry" so that for any calls to them, we can jump to the starts of these functions using these two variables.

- **Function prototypes without full definitions:** In C-, we can declare function prototypes, but no full definitions are provided later, we can't really generate the code for them. This can be identified by checking the body of a FunctionDec node. If it is an instance of NilExp, we can also report an error before terminating the code generation process.