

TM Simulator

CIS*4650 (Winter 2020)

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) );
```

Possible errors:

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

Register Only (RO) Instructions

Format: opcode r, s, t

<i>Opcode</i>	<i>Effect</i>
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)

<i>Opcode</i>	<i>Effect</i>
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

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 0 < r0 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

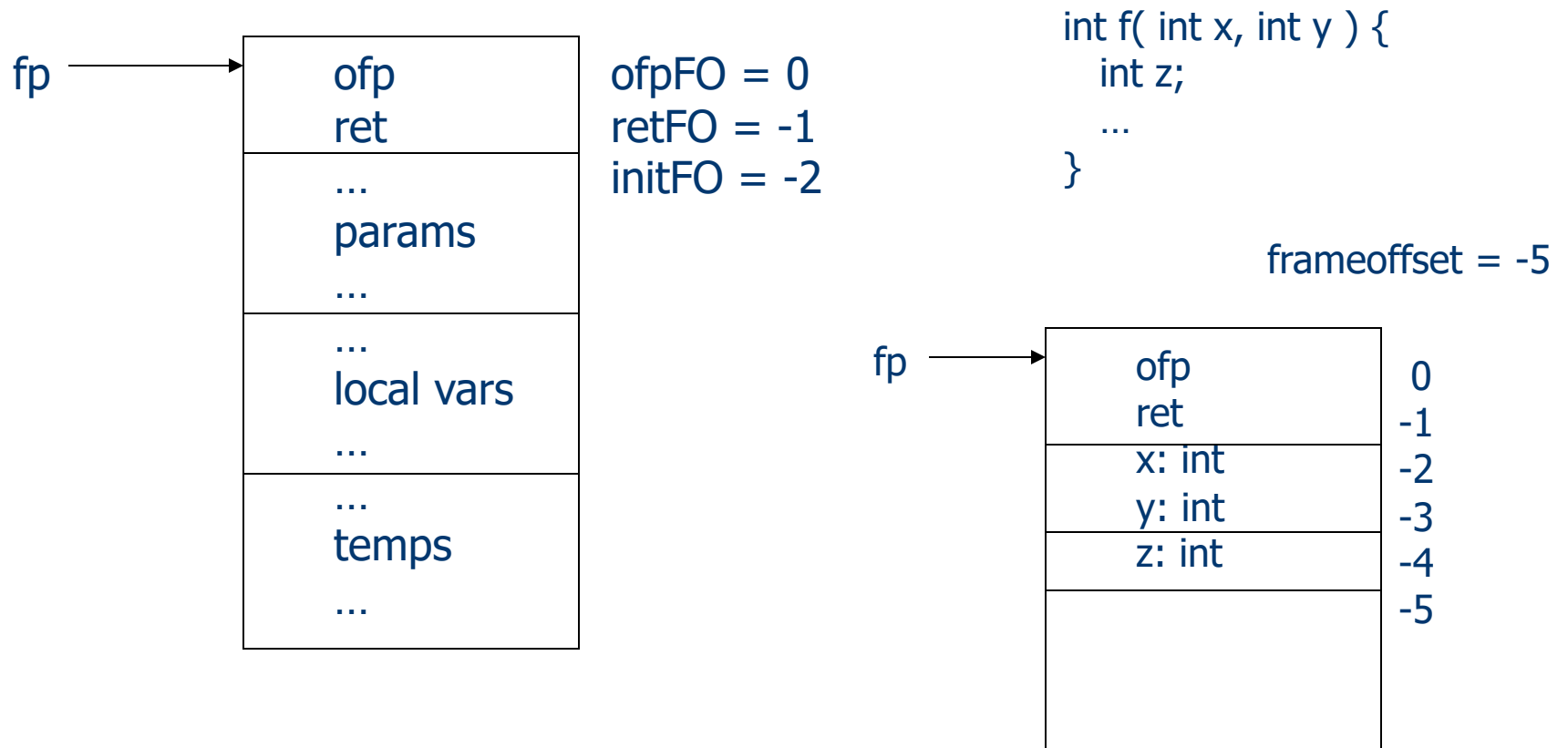
Flexible Order for Backpatching

```
0:    IN  0, 0, 0
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
1:    JLE 0, 6(7)
8:    HALT 0, 0, 0
```

skip an instruction

backpatching for a jump

Runtime Environment for C-Minus



Using Relative Addresses

➤ Accessing global data:

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

➤ Jump to a different instruction:

- e.g., from location 42 to location 27:

42: LDA pc, -16(pc)

Special registers:

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


Calling Sequence

● Call 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)

- * store current fp
- * push new frame
- * save return in ac
- * relative jump to function entry
- * pop current frame

● Return sequence:

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

Example Code

```
int y;
```

```
int gcd( int u, int v ) {  
    if( v == 0 )  
        return u;  
    else  
        return gcd( 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

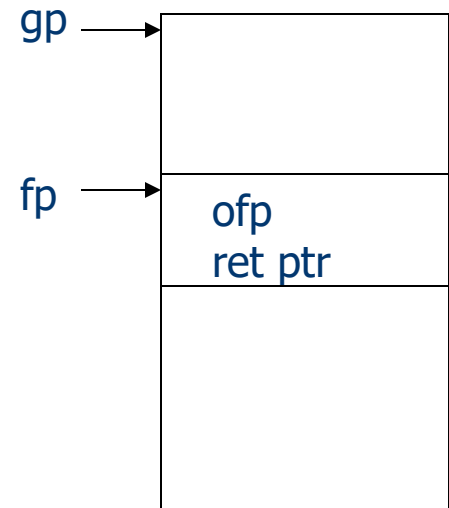
- finale:

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

Generating Finale

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

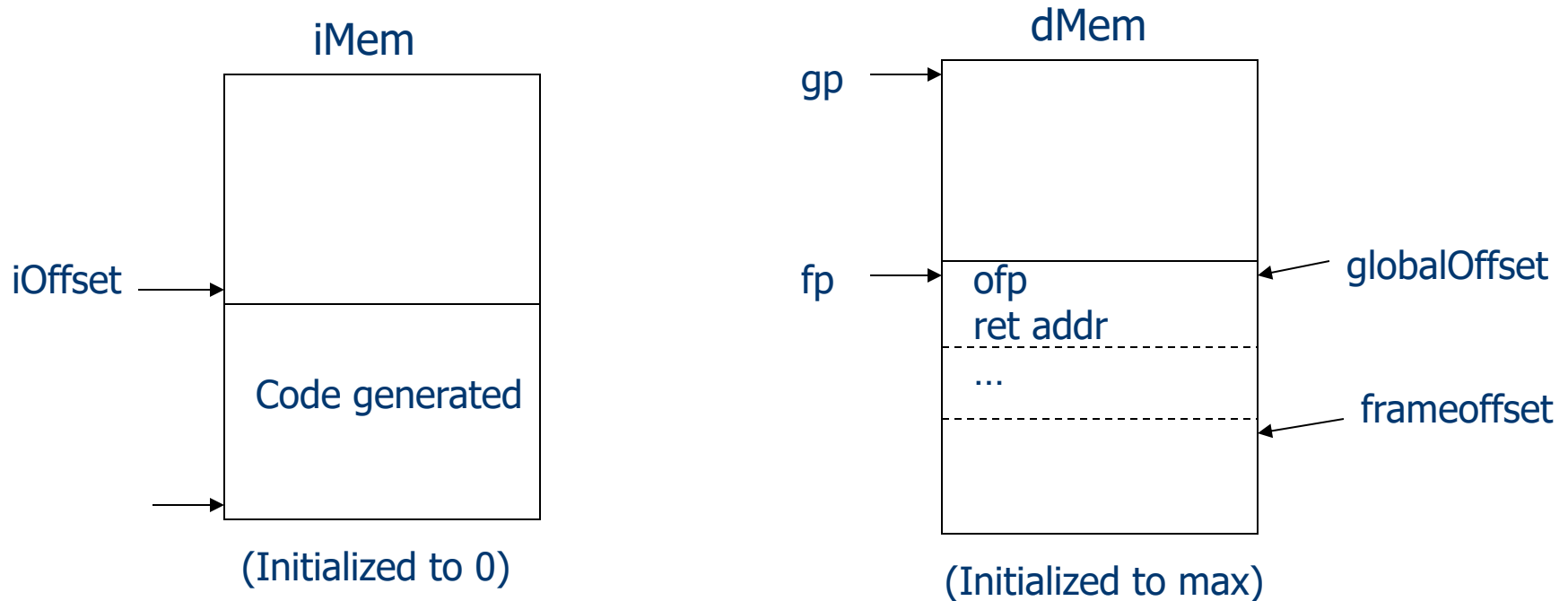
```
...
emitRM( "ST", fp, globalOffset+ofpFO, fp, "push ofp" );
emitRM( "LDA", fp, globalOffset, fp, "push frame" );
emitRM( "LDA", ac, 1, pc, "load ac with ret ptr" );
emitRM_Abs( "LDA", pc, entry, "jump to main loc" );
emitRM( "LD", fp, ofpFO, fp, "pop frame" );
emitRO( "HALT", 0, 0, 0, "" );
```



- Beginning of a function:

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

Three Offsets to Keep Track of



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

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

```
/* code for backpatching */
```

```
...
```

```
int savedLoc = emitSkip(1);
```

```
/*
```

```
    code for the branch
```

```
*/
```

```
int savedLoc2 = emitSkip(0);
```

```
emitBackup( savedLoc );
```

```
emitRM_Abs( "LDA", pc, savedLoc2, "" );
```

```
emitRestore();
```

```
...
```

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;  
}
```

```
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;  
}
```

Extending AST's for Declarations

- Four different data types: int, void, array, and function

array: size

function: funaddr (absolute start address in the code)

varDec: int, void, and array

offset: location relative to fp

nestLevel: gp or the current fp

- References:

ID: child[0] points to a varDec

Calls: child[0] points to a functionDec

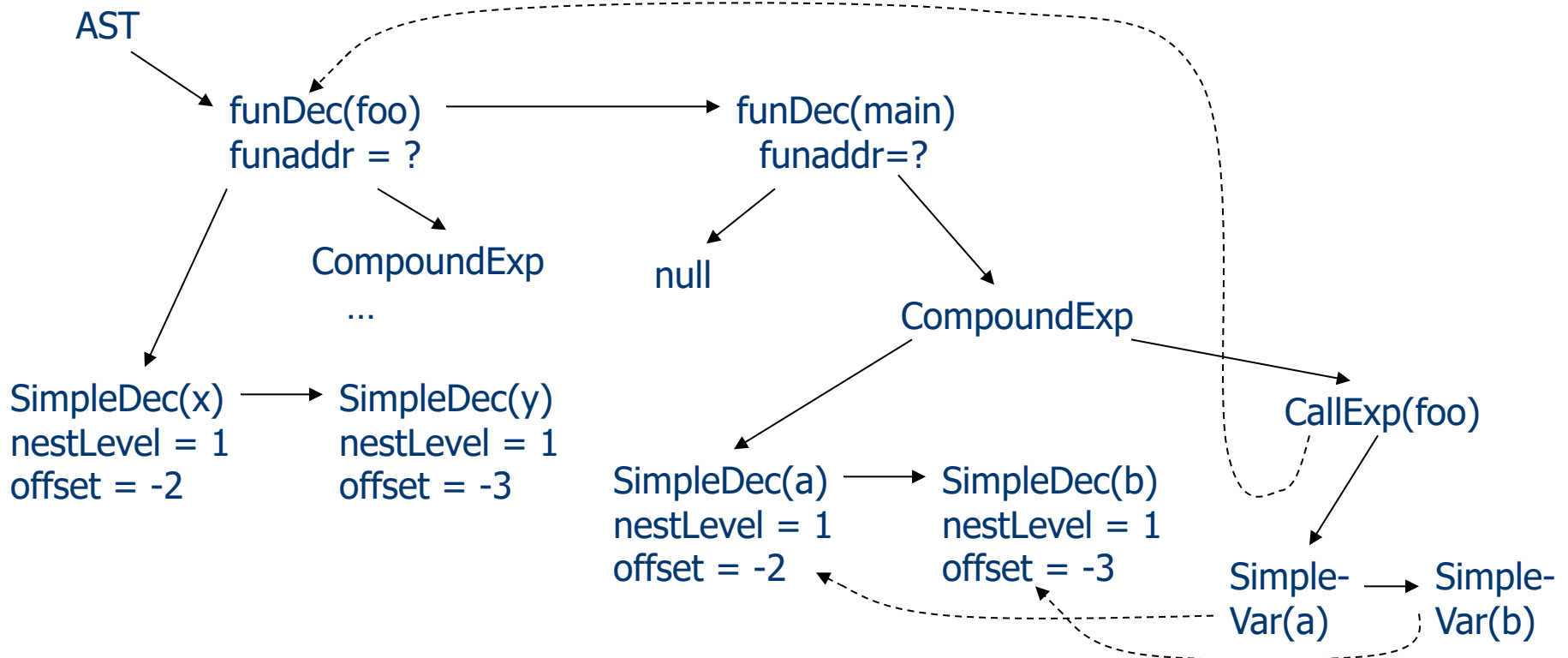
Extending AST's for Declarations

```
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



High-level Structure for CodeGen

```
void codeGen( Absyn tree ) {  
  
    /* generate standard prelude */  
    ...  
  
    /* generate i/o routines */  
    ...  
  
    /* generate code through the related methods in the  
       visitor class for different kinds of tree nodes */  
    cGen( tree,  
          /* offset = */ 0,  
          /* isAddress */ false );  
  
    /* generate finale */  
    ...  
}
```

Incremental Development (1)

(1) Extend AST's for memory locations:

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.

Incremental Development (2)

(2) Implement all the emit routines in your visitor class for code generation.

```
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;
}
```

- 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.

Incremental Development (3)

(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=-10".
- 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 in the stackframe.

Incremental Development (4)

(4) 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 */  
...
```

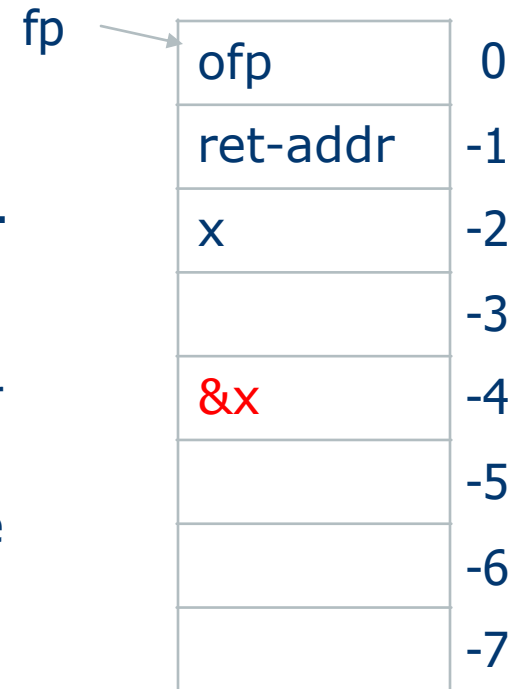
- After that, we can gradually add code generation functions for expressions, assignments, control structures, functions and recursions, nested blocks, arrays, and runtime error checking, as suggested in the marking scheme for Checkpoint Three.

Incremental Development (5)

(5) 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 “cGen(tree, -3, false)” where “tree” is an AssignExp and “-3” is the frameOffset.
- Inside the “cGen” for AssignExp, we will call “cGen(tree.lhs, -4, true)” and “cGen(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 “cGen” 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)



Incremental Development (6)

- Inside the "cGen" for OpExp, we will call "cGen(tree.left, -6, false)" and "cGen(tree.right, -7, false)" in the post-order traversal. The former is for a SimpleVar and the latter is for an IntExp.

- Inside the "cGen" 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)

fp → ofp 0

- Inside the "cGen" 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 "cGen" 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 "cGen" 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)

ofp	0
ret-addr	-1
x: result of x+3	-2
result of x + 3	-3
&x	-4
result of x+3	-5
value of x	-6
value of 3	-7

Incremental Development (7)

- Note that in the above illustration, we need to handle the “cGen” for SimpleVar differently depending on whether we are computing the left-hand side of AssignExp or not. This is distinguished by “isAddr” parameter in the “cGen(Absyn tree, int offset, boolean isAddr)”.
- The value for “isAddr” is false for most cases except when calling “cGen(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.

Incremental Development (8)

- 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.

