Lecture 5

Nested macro calls – 1/12

- Two basic alternatives exists for processing nested macro calls

+	MOVEM	BREG, TMP	
+	**INCR**	**X, Y, REG=BREG**	**--Nested macro**
+	MOVER	BREG, TMP	

Nested macro calls – 2/12

First alternative

The macro expansion schematic can be applied to first level expanded code to expand these macro calls and so on, until we obtain a code form which does not contain any macro calls

Require number of passes of macro expansion

Expensive

Nested macro calls – 3/12

Second and efficient alternative

Examine each statement generated during macro expansion to see if it is itself a macro. If it is a macro, expand this call before continuing with expansion of parent macro call

Avoids multiple passes of macro expansion

Ensures processing efficiency

Nested macro calls – 4/12

MACRO
INCR &A, &B, &REG=AREG
MOVER &REG, &A
ADD &REG, &B
MOVEM &REG, &A
MEND

MACRO
COMPUTE &F, &S
MOVEM BREG, T
INCR &F, &S, REG=BREG
MOVER BREG, T
MEND
The macro call is COMPUTE X, Y

Nested macro calls – 5/12

- When ADD statement is being generated
- **Expansion of two macro calls is in progress at this moment**
- Happened because outer macro COMPUTE gave rise to a macro call INCR during expansion of its current model statement
- Model statements of INCR are currently being expanded using expansion time data structures MEC, APTAB, EVTAB, APTAB_ptr and EVTAB_ptr

Nested macro calls – 6/12

- MEND statement encountered during expansion must lead to resumption of expansion of outer macro

- Requires that MEC, APTAB, EVTAB, APTAB_ptr and EVTAB_ptr should be restored to values contained in them while macro COMPUTE was being expanded

- **Control returns to processing of source program when MEND statement is encountered during processing of COMPUTE macro**

Nested macro calls – 7/12

- Two provisions required
  1. Each macro under expansion must have its own set of data structures
  2. Expansion nesting counter (Nest_ctr) is maintained to count the no of nested macro calls. It is incremented when macro call is recognized and decremented when MEND stmt is encountered.
     - Nest_ctr >1 means nested macro call is under expansion
     - Nest_ctr = 0 means macro expansion is not in progress currently

Nested macro calls – 8/12

- **First provision** can be implemented by creating many copies of expansion time data structures
- **They can be stored in form of array**
- Can have an array APTAB_ARRAY, each element of which is APTAB
- APTAB for innermost call would be given by APTAB_ARRAY[Nest_ctr]

- Provides access efficiency
- Expensive in terms of memory requirements

Nested macro calls – 9/12

- Since macro calls are expanded in LIFO manner, a **stack is used to accommodate the expansion time data structures**
- Stack consists of expansion records
- Each expansion record accommodating one set of expansion time data structures
- **Expansion record at top of stack corresponds to macro call currently being expanded**

Nested macro calls – 10/12

- When nested macro call is recognized, new expansion record is pushed into stack to hold data structures for the call
- At MEND, expansion record is popped off the stack

- **Expansion record on top of stack contains the data structures in current use**
- Record base (RB) is pointer pointing to start of this expansion record
- TOS points to last occupied entry in stack

Nested macro calls – 11/12

- When a nested macro call is detected, another set of data structures is allocated on stack
- RB is now set to point to start of new expansion record
- MEC, EVTAB_ptr, APTAB and EVTAB are allocated on the stack in that order

- During macro expansion, the various data structures are accessed with reference to value contained in RB

Nested macro calls – 12/12

- **At MEND stmt, record is popped off the stack by setting TOS to end of previous record**
- Necessary to set RB to point to start of previous record in stack which is achieved by using entry marked ‘reserved pointer’ in expansion record
- This entry always point to start of previous expansion record in stack
- While popping off a record, value contained in this entry can be loaded into RB

Design of a Macro assembler (Please Go Thro By Yourself)

- Use of macro preprocessor followed by assembler is expensive way of handling macros since the no of passes over source program is large and many functions get duplicated
- Eg. Analysis of source statement to detect macro calls requires us to process the mnemonic field
- Similar function is required in the first pass of assembler
- Similar functions of preprocessor and assembler can be merged if macros are handled by a macro assembler which performs macro expansion and program assembly simultaneously
- Also reduce no of passes

Pass structure of macro-assembler – 1/4

- Identify functions of a macro preprocessor and assembler which can be merged to advantage
- After merging, functions can be structured into passes of the macro-assembler
- Process leads to following pass structure
  - Pass I
    - Macro definition processing
    - SYMTAB construction

Pass structure of macro-assembler – 2/4

- Pass II
  - Macro expansion
  - Memory allocation and LC processing
  - Processing of literals
  - Intermediate code generation
- Pass III
  - Target code generation

Pass structure of macro-assembler – 3/4

- Pass II is large in size since it performs many functions
- Also, since it performs macro expansion as well as Pass I of assembler, all data structures of macro preprocessor and assembler need to exist during this pass
- The pass structure can be simplified if attributes of actual parameters are not to be supported
- Macro preprocessor would then be a single pass program

Pass structure of macro-assembler – 4/4

- Integrating Pass I of assembler with the preprocessor would give us the following two pass structure
  - Pass I
    - Macro definition processing
    - Macro expansion
    - Memory allocation, LC processing and SYMTAB construction
    - Processing of literals
    - Intermediate code generation
  - Pass II
    - Target code generation