Assemblers

- Two functions:
  - Mnemonic opcode ⎵ Machine Code
  - Symbolic labels ⎵ machine addresses (in memory)
- Some features:
  - Depend on assembly language
  - and machine language as well, may be very subtle.
- Other features:
  - Machine-independent,
  - Even are arbitrary decisions by the designers of the language
- Follows the steps for chapters.

An assembly language program as example

- Source Code
  - Normally, when entering into subroutine/method/function/procedure, have STL to store the return address in Linkage register into a variable, before return using RSUB, have LDL to load return address (from the variable) into Linkage register, particularly when it calls other subroutine(s) within it to avoid the return address in Linkage register to be over-written by the new return address.
  - Even subroutine call JSUB and return RSUB are hardware-supported, but they are parameter-less. Parameters/arguments of a method/function/procedure are software features, and need to be specifically copied with/passed when you write assembly/machine language code yourself or by compiler.
- Mnemonic instructions
- Symbolic labels
- Assembler directives
- Function
  - Read from an INPUT device (code F1) and then copy to an OUTPUT device (code 05)
  - Call two subroutines: RDREC and WRREC
  - Called from OS by JSUB, so save return address and return by RSUB
  - Support buffer size of 4096 and no error check.

A Simple SIC Assembler

- Translation of source code to object code needs:
  1. Mnemonic opcodes to equivalent machine codes
     1. E.g, STL to 14
  2. Symbolic labels to equivalent machine address
     1. E.g. RETADR to 1033
  3. Build machine instructions in proper format.
  4. Convert data constants into internal machine representation, such as EOF to 454F46
  5. Write the object program and the assembly listing.

Object Program

- Program from Fig2.1 with Object Code (Fig.2.2)
- Comments:
  - Instructions and data are all in binary
  - A same value can be an instruction or data, or address
  - They are mixed together/interleave.
  - Object code is written into a file with certain format and then loaded into memory for execution later by a loader
- Object Program corresponding to Figure 2.2 (Fig2.3)
  - Three kinds of records: H—header, T: text, E: end.

Problem and solution

- Symbolic label and forward reference
  - A reference to a label which is defined later.
  - How to solve it?
- Two passes
  - Scan label definitions and assign addresses
  - Do actual translations
- Other functions:
  - Process assembler directives such as START, END, BYTE, WORD, RESB, RESW, …
  - Write the object code onto output devices.
    - Three kinds of records: Header, Text, and End (in hexadecimal)
    - In the future, Modification record, Define record, Refer record,…

General Functions of an assembler

- Pass one
  - Assign addresses to all statements in source code
  - Save values (addresses) assigned to labels for use in pass two
  - Process directives
- Pass two
  - Translate instructions
  - Convert labels to addresses
  - Generate values defined by BYTE and WORD
  - Process the directives not done in pass one
  - Write object code to output device

Assembler algorithm and Data Structure

- Two major internal data structures
  - OPTABLE:
    - (mnemonic opcode, machine code, instruction format, length)
    - Look up and validation in pass one, also increase LOCCTR by length
    - Translate opcode to machine code (including format) in pass two
    - Organized as a hash table.
  - SYMTAB
    - (label name, address, flags, other info such data type or length)
    - Pass one: labels are entered along with their addresses (from LOCCTR).
    - Pass two: labels are looked up to get addresses to insert in assembled instructions
    - Organized as a hash table too.
- Location Counter Variable: LOCCTR
  - Memory locations (addresses)
  - Initialized as the address in START
  - Increased by adding the #bytes of an instruction or data area
  - When encountering a label, the current LOCCTR gives the address of the label.

Connection between two passes

- Pass one writes an intermediate file,
  - Source instructions along with addresses, flags, pointers to OPTAB and SYMTAB, and some results from the first pass

A simple Two pass assembler

- Algorithm for pass 1 of an assembler (Fig2.4(a))
- Algorithm for Pass 2 of an assembler (Fig2.4(b))
- Very important for you to understand the programs and to be able to implement an assembler.

Machine-dependent Assembler Features

- Example of a SIC/XE program (Fig2.5)
- Real machines have features similar to those of this SIC/XE, so the discussion and understanding of this SIC/XE will apply to other machines.
- New features
  - Indirect addressing @, e.g. line 70
  - Immediate addressing #, e.g., lines 25,55,133
  - extended instruction (4-byte) +, e.g., lines 15, 35, 65
  - Memory references are normally assembled as PC-relative or base-relative (with PC-relative first).
    - So BASE assembler directive (line 13).

Differences of SIC and SIC/XE programs

- Register-to-register instructions
  - COMP ZERO to COMP A,S
  - TIX MAXLEN to TIXR T
  - faster
- Immediate operands
  - Faster
- Indirect addressing
  - Avoid another instruction
- Some additions of instructions
  - COMP to COMPR (line 50), with addition of CLEAR instruction
  - But still faster, since CLEAR is executed once for a record but COMPR is executed once for each byte.
- So changes to assembler
  - Instruction formats and addressing modes
  - Multiprogramming and program relocation.

Instruction formats and addressing modes

- Program of Figure 2.5 with object code (Figure 2.6)
- Start address: 0
  - Translated exactly as if it is loaded to address 0
  - But the program should run anywhere it is loaded in memory.
- Register to register instruction:
  - no problem, just Register mnemonics ␣ numbers (in Pass 2), with a separate table or SYMTAB.
- Register to memory
  - PC relative or Base relative
  - Assembler must compute disp (with 12 bits) to put in object code
    - 10 0000 FIRST STL RETADR 17202D
      - The length of this instruction is 3.
      - PC relative, (PC=3), and RETADR is 0030 (line 95, from SYMTAB), so 30-3 =2D, (so 02D is disp).
      - During execution, (PC)+02D =0030, which is the address of RETADR.
    - Another example: 40 0017 J CLOOP 3F2FEC
      - Disp=address –(PC) =6 -001A=-14, in 2's complement, it is FEC.
    - 12 0003 LDB #LENGTH 69202D
      - Disp=target_address-(PC)=0033 – 06= 002D
      - Immediate addressing
      - Thus, (n,i,x,b,p,e)=(0,1,0,0,1,0). LDB's opcode is 68.
      - So, 0110 10 01 0010 0000 0010 1101, which is 69202D.
      - The combination of PC relative and immediate addressing.

Instruction formats and addressing modes

- Program of Figure 2.5 with object code (Figure 2.6)
- Register to memory
  - PC relative or Base relative
    - 160 104E STCH BUFFER, X 57C003
      - Using PC relative first, (PC)=1051, Address of BUFFER=0036, which is target address.
      - disp= TA - (PC)=0036-1051 = -101B, which cannot fit into 12 bits.
      - Thus, PC relative does not work, use Base relative instead.
        - » What is BASE register value so that the assembler can use it to compute disp?
          - Line 13 BASE LENGTH , an assembler directive , tells the assembler that B register contains the address of LENGTH, i.e., 0033
        - » But how does the CPU get (to know) the base value to compute target address?
          - Line 12, LDB #LENGTH, which is an instruction to load the address of LENGTH into base register B
          - the content of BASE is under the control of the programmer, not the assembler.
        - » As a result, LDB and BASE must use together.
      - BASE, other BASE, NOBASE to change, but no object code is generated.
      - So disp = address of BUFFER – (Base, i.e., the address of LENGTH) = 36-33 =003. in addition, ,X for index addressing. So x and b are both set to 1.
    - Another example Line 175 1056 EXIT STX LENGTH 134000
      - (Base relative, disp=0.)
    - Please try 225 1068 LDCH BUFFER,X 53C003
    - Line 20, 000A LDA LENGTH 032026 (PC based.),
      - is it fine to use BASE relative?

Instruction formats and addressing modes (cnt.)

- Register to memory
  - Rules:
    - Always use PC relative first as default.
    - If disp is out of 12 bit limit, use base relative.
    - If disp (in base relative) is out of 12 bit limit, report error, unless the instruction is in extended format indicated by +.
    - For extended format (mode 4)
      - So full address, relative addressing is not needed.
      - Clearly indicate by programmer, such as +, otherwise, an error is reported. (How to processed?)
      - Line 15, +JSUB RDREC, the address of RDREC is 1036 which is directly put in 20 bits of the instruction.
      - Bit e is set to 1.
        - » 4B101036
- Immediate addressing: easy,
  - Line 55
  - Line 133, 4096 is too large to fit into 12 bit disp, so extended format.
  - Line 12, immediate operand is the *symbol* of LENGTH, i.e., the address of LENGTH.
    - Also PC relative. So Disp =address of LENGTH –(PC)=33-6=2D.
    - So TA= (PC)+ disp =33 as operand directly, loaded into B.
  - Line 55, all x,b,p are set 0.
- Indirect address: disp is computed as usual, and bit n is set.
  - Line 70: PC relative and indirect addressing.
- Address and data (including constant items) are not distinguishable.

Program relocation

- Multiple programs, not sure what and when.
- Load a program into where there is a room
- So actual starting location is not known until load time.
  - Absolute assembly will not work.
  - Also, the assembler cannot change or set actual address.
  - However, the assembler can tell the loader which addresses need to be modified into actual locations
  - So re-locatable assembly.
- An object program containing necessary modification information is called re-locatable program.
- Examples of program relocation (Figure 2.7)
  - No matter where the program is loaded, RDREC is always 1036 bytes pass the starting address of the program.

Ways for program relocation

- Assembler generates and inserts the relative address of RDREC (to the start of the program) into the object code
- Assembler will also produce a command for the loader, instructing it to add the beginning address of the program into the instruction.
  - Which is part of the object code,
  - So Modification record.
    - Col.1 M
    - Col. 2-7: starting (relative) location of address field to be modified
    - Col. 8-9: length of the address field (in half byte)
  - So JSUB Modification record: M00000705
  - Line 35 and 65: M00001405, M00002705
- Many instructions do not need modification
  - Immediate addressing
  - PC relative or Base relative
- Only direct addresses:
  - In SIC/XE: just extended format (4-byte instructions).
  - One advantage of relative address.
- Object program corresponding to Figure 2.6 (Figure 2.8)
- What does a loader do?

Machine independent assembly features

- Literals (Sec. 2.3.1)
- Typical assembler directives: EQU, ORG (Sec. 2.3.2)
- Expressions (Sec. 2.3.3)
- Program blocks and control sections (Sec. 2.3.4 and 5)

Literals

- A constant value in an *assembly* instruction directly as an operand without a need of a label.
- Denoted by =, e.g., line 45 and 215
- Program demonstrating additional assembler features (Figure 2.9)
- Difference between # and =:
  - With #, operand value is assembled as part of the *machine* instruction.
  - With =, the assembler generates the specified value as a constant in an other location of the memory and include the address of the constant as TA in the *machine* instruction.
- Free programmers from defining the constant, instead, the programmer uses the value directly.
- Literal pools: one or more pools. One is at the end of the program.
- LTORG directive: --line 93.
  - Indicate the location of literals encountered from the last LTORG or the beginning of the program.
  - Avoid too far away of the literal constants from its referring locations, which may be the case when putting all literals at the end of the program.
- Program from Figure 2.9 with object code (Figure 2.10)
  - Object codes for lines 45 and 215 are identical to the corresponding codes in Figure 2.6.
  - Look at line 45 and 55 to understand the difference of # and =.
- Duplicate literals:
  - allowed but the assembler just generates one constant in memory.
  - Different definitions but the same value: such as =C'EOF' and =X'454F46',
    - The assembler can deal with it but it may not worth.
  - Same name but different values: such as *, indicating the current value of location counter.
    - Programmer needs to pay much attention
- How does the assembler process literals and what is the data structure needed?

Processing literals

- Data structure: LITTAB,
  - Name, value, length, and address
  - preferred a hash table, with literal name/value as key
- In Pass 1:
  - When encountering a literal, search LITTAB via name/value, if not found, insert (name, value, length) into LITTAB
  - When LTORG or end of the program is encountered, scan LITTAB and assign addresses, modify location counter accordingly.
- In Pass 2:
  - For each literal encountered, search LITTAB, get the address and insert into the machine instruction.
  - Moreover, the value of the literal is inserted into an appropriate location in the object program.
  - If the literal value represents an address, (e.g., a program location counter), must generate proper Modification record.

Symbol definition statements

- Similar to constant definition in C/C++
- For example: +LDT #4096
  - MAXLEN EQU 4096
  - +LDT #MAXLEN
- Use symbolic name for a numeric value to improve readability and also easy modification.
- Defining mnemonic names for registers.
  - A EQU 0, X EQU 1, L EQU 2
- Define the usage of general purpose registers
  - BASE EQU R1, COUNT EQU R2, INDEX EUQ R3
- How to process EQU by assembler?
  - Put into SYMTAB when the definition is encountered,
  - Search SYMTAB for the value during its usage of an instruction.
- Another directive is ORG. (Omitted, look at it if interested).
- Note: define a symbol first and then use it.

Expressions

- Operators plus terms well formed
  - constants, user-defined symbols, or special terms.
  - * (current value of location counter) is a specific term, e.g. BUFEND EQU *, so BUFEND will be the address of the next byte after the buffer.
- The value of a term or an expression can be either absolute or relative.
- An expression must be able to evaluate to a single value.
- Two relative terms that can be paired will generate a absolute value.
  - E.g., MAXLEN EQU BUFEND-BUFFER
- Assembler must have the ability to compute well-formed expressions, otherwise, report error.
- In order to indicate relative or absolute, SYMTAB must contain a column TYPE.
- For absolute expressions, all the relative terms must be paired.
- For relative expressions, all except one must be paired.

Program Blocks

- Previously, A single block of object code: so the instructions and data in object code appear in the same order as in source program.
- Program blocks: segments of code that are rearranged within a single object program unit.
- Control Sections: segments that are translated into independent object program units.
- Example of a program with multiple program blocks (Figure 2.11)
  - first (unnamed) block—executable code
  - CDATA block—data areas of short length
  - CBLKS block – data areas of large length
- USE directive
  - USE <name> begins a new block <name> or resume a previous block <name>
  - USE resume the default (unnamed) block.
  - A block can contain separate segments of code in source program. Assembler will logically re-arrange them together.
  - Free programmer from re-arranging/sorting the segments.
- How does the assembler process program blocks?

Processing program blocks

- A program block table: block name, block number, address, length, and *program block location counter*.
- In Pass 1,
  - Location counter is initialized to 0 when the block begins.
  - Save the location counter value when switching to another block and restore the location value when resuming a previous block.
  - Each label is assigned the address which is relative to the start of its block. The block name or number the label belongs to is also stored along with its relative address.
  - At the end of Pass 1, the location counter of each block indicates the length of the block.
  - Finally, each block is assigned a start address which is relative to the beginning of the entire program (generally 0).
- In Pass 2,
  - Compute the address for each symbol, which is relative to the start of entire object program
    - i.e., the relative starting address of the block + the relative address of the symbol (in the block)
- Three program blocks:
  - Default, 0, 0000, 0066 *(,block location counter is useless for pass 2)*; CDATA, 1, 0066, 000B; CBLKS, 2, 0071, 1000
- Example: 20 0006 0 LDA LENGTH 032060
  - LENGTH: 03 (relative to CDATA block, which has relative address 66),
  - So the relative address (to the start of the program) is 66+03=69.
  - Using PC relative, when the instruction is executed, the PC will be 0009, which is relative to the start of this block. So the PC, relative to the start of the entire program, will be 0009+0000=0009.
  - So the disp will be 69 –(PC)= 60.
- Advantages:
  - Reduce addressing problem, without considering large data and reducing extended instruction format (e.g, line 15,35,65).
  - Base registers can be removed (such as LDB and BASE)
  - Literal placement is easier: just include LTORG in CDATA to make sure the literals are places ahead of large data.
- One disadvantage:
  - Readability is better if the data is closed to the instructions which refer to them.
  - Programmer can insert data anywhere in the program (of course, need jump around). THIS IS REAL PROGRAM!!
  - But blocks separates the data with their instructions.

Processing program blocks (cont.)

- It is not needed to place the pieces of each block together in object program.
- Instead, the assembler simply write the object code as it is generated in Pass 2, and insert the proper load address in each Text record.
- It does not matter that the Text records are not in sequence by addresses, the loader can just load the records into the indicated addresses.
- Object Program corresponding to Fig 2.11 (Figure 2.13)
  - Two Text records for default block
  - CDATA(1) and CBLKS do not generate object code.
  - Two more Text records for default block
  - One Text record for CDATA(2) (one byte)
  - Two more Text records for default block
  - One Text record for CDATA(3) (4 bytes)
- Program blocks from Fig2.11 traced through assembly and loading processes (Figure 2.14)
- As it can been seen:
  - The pieces of a block are in different places of object program
  - Some data blocks may not appear in object program, e.g. CDATA(1), CBLKS(1)
  - After loading, the pieces of a block will be placed together and
  - The memory will be automatically reserved for data blocks , even they do not appear in object program.
- Examine Fig2.12, 2.13, and 2.14 carefully to understand how the assembler (along with loader) deals with program blocks.
- Also get some idea about loaders.

Control sections and Program Linking

- A control section can be processed Independently.
- Main advantage: flexibility
- When control sections form logically related parts of a program, some linking method is needed.
- Reference between control sections are called external reference.
- Assembler generates information for each external reference to allow the loader to perform required linking.

Control sections

- Default control section
- CSECT directive, indicate the beginning of a new CSECT.
- Different control sections may not be in one source program.
- EXTDEF: define a symbol as an external which can be used in other control sections
- EXTREF: indicate an external reference of a symbol which is defined in another control section.
- Illustration of control sections and linking
- How to deal with by assembler?

Dealing with control sections

- Two more record: Define record and Refer Record
- For external definitions, add Define records in object program
- For external reference, add Refer records in object program
- Also change Modification record by adding:
  - Modification flag: + or –
  - External reference name
- When each external reference is encountered,
  - ZERO is set as its address or is used to compute the value of an expression
  - No relative addressing, so extended format is used.
  - If relative references, must be paired and the paired two symbols belong to the same control section.
- Must clearly indicate external symbols, otherwise, assembler will give errors to the symbols not defined.
- Program from Figure 2.15 with object code (Figure 2.16)
- 15 0003 CLOOP +JSUB RDREC 4B100000
  - RDREC is an external reference, assembler does not know where it is,
  - so address ZERO is inserted to the instruction.
  - Also no relative addressing, so extended format
  - Modification record is added in the object code: M00000405+RDREC
  - When loaded, the modification record will allow the loader/linker to add RDREC's address into the address.
- 160 0017 +STCH BUFFER,X 57900000
  - BUFFER—external ref, so ZERO as address, ,X is index addressing, so x bit is set to 1
  - Modification record M00001805+BUFFER is added in the object code
  - When loaded, this modification record allows the loader/linker to add the address of BUFFER to the instruction.
- 190 0028 MAXLEN WORD BUFEND-BUFFER 000000
  - Both are external references, so 00000 is set as the value of MAXLEN
  - But two modification records is added in the object program of CSECT RDREC
    - M00002806+BUFEND M00002806-BUFFER
  - When the object is loaded, these two records will allow the loader or linker to add/sub correct values
- Compare: 107 MAXLEN EUQ BUFEND-BUFFER
  - Since MAXLEN , BUFEND, and BUFFER are in the same control section, so can be computed by the assembler directly.

Control sections

- Define record:
  - Col. 1: D
  - Col.2-7: Name of the external symbols defined in this section
  - Col.8-13: Relative address in this section
  - Col. 14-73: repeated info. Of Col.2-13 for others
- Refer record:
  - Col. 1: R
  - Col.2-7: Name of external symbol referred to in this section
  - Col.8-73: Names of other external reference symbols.
- Modification record:
  - Col. 1: M
  - Col.2-7: Starting address of the field to be modified
  - Col. 8-9: length of the field to be modified
  - Col. 10: modification flag (+ or -)
  - Col.11-16: external symbol whose value is to be added or subtracted.
- Object program corresponding to Fig2.15 (Figure 2.17)
- The revised Modification record can also be used for program relocation:
  - From Figure 2.8: M00000705 ␣M00000705+COPY (Here COPY, as a control section, has as its value the required address, i.e., starting address when loaded)
- Expressions with external references:
  - Paired terms must be with the same control section.
  - Assembler cannot determine the external terms for evaluation, so Modification record is generated so that the loader can finish evaluation when loading the codes.

Assembler Design Options --one pass assembler

- Forward reference
  - Data: just put data before the codes which refer the data so prohibit forward reference
  - But how about forward jump? Prohibiting forward jump is too restrictive and inconvenient.
- Sample program for one-pass assembler (Figure 2.18)
- Two scenarios:
  - Generate object code in memory for immediate execution
    - Where to use it?
      - For development and testing.
    - Efficiency is main concern.
    - Also easier, since no need to write out.
    - How to do:
      - Symbol, flag, its forward reference list.
      - When the definition of a symbol is encountered, scan the list to insert its address into all the instructions on the list.
    - Object code and symbol table after Scanning line 40 (Figure 2.19(a))
    - Object code and symbol table after scanning line 160 (Figure 2.19(b))
  - Write the object code into file for later executions
    - When to use it?
      - Working device for immediate file is not available, slow, or inconvenient.
    - How to do?
      - Symbol table is same as before.
      - But cannot go back to modify an instruction since its Text record may write out on file already
      - Additional Text record is generated, such as the third record: T00201C 02 2024
      - So leave the task to loader.
    - Object program for one-pass assembler with addition Text record (Figure 2.20)
  - We just consider single symbol and actual (not relative) address.
    - How about other features, such as literals and expressions?

Assembler Design Options ---multiple pass assembler

- Recall for EQU statement:
  - A symbol on the right-hand of an EQU must be define previously.
  - It is a basic requirement for all 2-pass assemblers.
- Consider the following example:
  - ALPHA EQU BETA
  - BETA EQU DELTA
  - DELTA RESW 1
- BETA cannot be assigned value during pass 1 and ALPHA cannot be assigned during pass 2, so more passes are needed.
  - Forward reference is bad for both programmer to read and for machine to process.
  - If more than two passes, not entire source code to be scanned, but just portion of the code.
- Solutions
  - Store the definition of a symbol involving forward references.
  - Also a dependent list: record the symbols whose values are dependent on the symbol
- A few figures.
  - Example of multiple assembler operation (Figure 2.21) (a)) (b)) (c)) (d)) (e)) (f)

Implementation examples --MASM assembler for Pentium

- A collection of segments.
  - Each segment belongs to a specific class
    - Common classes: CODE, DATA, CONST, STACK
  - Segments are addressed by segment registers:
    - Segment registers are automatically set by loader.
    - CODE: CS,
      - is set to the segment containing the starting label specified in the END statement.
    - STACK: SS
      - Is set to the last stack segment processed by the loader.
    - DATA: DS, ES,FS,GS
      - Can be specified by programmers in their programs.
      - Otherwise, one of them is selected by assembler.
      - DS is the data segment register by default
        - » Can be changed and set by: ASSUME ES:DATASEG2
        - » Any reference to the labels defined in DATASEG2 will be assembled based on ES
      - Must be loaded by program before they can be used.
        - » MOV AX,DATASEG2
        - » MOV ES, AX
      - ASSUME is somewhat similar with BASE in SIC, programmer must provide instructions to load the value to registers.
    - Collect several segments into a group and use ASSUME to link a register with the group.
  - Parts of a segment can be separated and assembler arranges them together, like program blocks in SIC/XE

Implementation examples --MASM assembler for Pentium (cont.)

- JMP is a main specific issue:
  - Near JMP: 2 to 3 bytes, same segment, using current CS
  - Far JMP: 5 bytes, different segment, using a different segment register, as the instruction prefix.
  - Forward JMP: e.g., JMP TARGET,
    - Assembler does not know whether it is a near jump or far jump, so not sure how many bytes to reserve for the instruction.
    - By default, assembler assumes a forward jump is near jump. Otherwise,
    - JMP FAR PTR TARGET , indicate a jump to a different segment
      - Without FAR PTR, error will occur.
      - Similar to SIC/EX extended format instructions.
    - JMP SHORT TARGET, indicate a within-128 offset jump.
- Other situations that the length of an instruction depends on operands. So more complicate than SIC/EX
  - Must analyze operands, in addition to opcode
  - Opcode table is more complex.
- References between segments that are assembled together can be processed by assembler
- Otherwise, it must be processed by loader.
  - PUBLIC is similar to EXTDEF
  - EXTRN is similar to EXTREF
- Object programs from MASM can have different formats.

Implementation examples --SPARC assembler

- Sections: .TEXT, .DATA, .RODATA, .BSS, and others
- Programmers can switch between sections and assemblers will re-arrange a section together– similar to SIC/XE blocks
- Local symbols among sections with a program and global symbols among sections in different programs, processed by linker.
  - Global or weak (weak symbols can be overridden)
  - Object program contains these information for linking.
  - Not similar to SIC/XE program blocks, but control sections.
- Delayed branch
  - The instruction following a branch instruction is actually executed before the branch is taken. Called delay instruction. Why?
  - The specific pipeline architecture
  - Put some useful or no harm instruction, or a NOP instruction
  - Annulled: the delay instruction is executed if the branch is taken and not if not taken. ,A is included in the branch instruction.

Implementation examples --AIX assembler for PowerPC

- Different modes of PowerPC machines
  - .MACHINE assembler directive.
- Any general purpose register (except GPR0) can be used as base register
  - .USING and DROP, similar to SIC/XE BASE
- Allow programmers to write base register and disp explicitly in the source program.
  - L 2,8(4): operand address is 8 bytes pass the address contained in GPR4.
- Control sections
  - Include both SIC/XE control sections and program blocks.
  - A control section can contain several program blocks.
  - *Dummy* section and *common* blocks
  - .GLOBAL and .EXTERN (similar to SIC/XE EXTDEF and EXTREF).
  - TOC (Tables of Contents) , generated by assembler and used by linkers
- Two-pass assembling:
  - Pass one also generates warning and error message
  - Pass two reads the source program again, not from the intermediate file.

Summary

- Functions
  - Basic
    - Mnemonic ␣ Opcode
    - Labels ␣ Addresses
  - Extended
    - Program relocation
    - Different instruction formats, addressing modes.
    - Literals, EQU, Expression, Blocks, Control Sections.
- Two passes:
  - Pass one
    - Assign addresses to all statements in source code
    - Enter names of symbols (labels/literals/blocks/Control Sections) into SYMTAB, LITTAB, ESTAB
    - Save values (addresses, lengths) assigned to symbols for use in pass two
    - Process directives
  - Pass two
    - Translate instructions
    - Convert symbols to addresses.
    - Generate values defined by BYTE and WORD and compute expressions to values.
    - Write object code to object program including Define, Refer, and Modification records.
- Data structures
  - Tables: OPTABLE, SYMTAB, LITTAB, ESTAB, … , hash/or linked list
  - Location counter: LOCCTR, PROGADDR, CSADDR, …
  - Reference list/dependent list
- Directives:
  - BYTE, WORD, RESB, RESW, BASE, EQU, USE, LTORG, CSEC, EXTDEF, EXTREF,
- Object records:
  - Header, End, Text, Modification, Define, Refer, …
  - Modification record: program relocation and external reference
  - Additional text record in single pass assembler
- Relations among source program, (intermediate file), object code, and object program.
- Relations among assembler, loader, and linker.