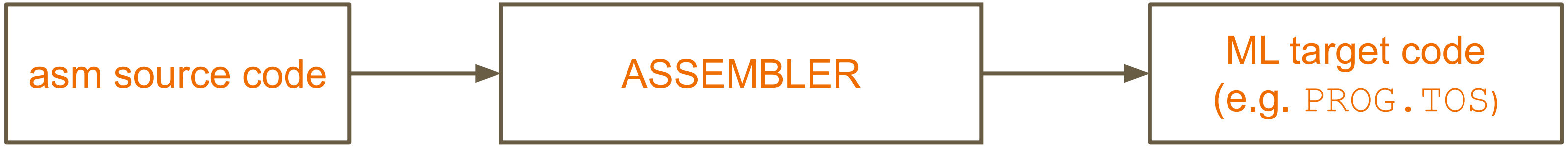
Assembler, Linker, & Loader

Assembler

is a program that translates an assembly language source file into object code



The Assembler

- Evaluates assemble-time expressions and expands macros
- Checks for errors
- Performs the actual translation

Macros are beyond the scope of the course, but can be found on page 110 in the Devpac manual.

```
Label macro
	<string>
	endm
```

Simply a string replacement process

Translation Process

- An assembler reads the source sequentially. This is called a pass.
- To handle forward and backward references, assemblers are often two pass assemblers:
- The assembler maintains a location counter.
- Pass 1 uses the location counter to build a symbol table:
- At the end of Pass 1, either all symbols are fully defined or an error is reported.
- Pass 2 performs the translation, using the symbol table to look up label locations.

Hand Assemble Using a Two-Pass Method

```
NULL     equu     0
start:        clr.w   d0
              lea     str,a0
              lea     copy,a1
copy_count: move.b  (a0)+,(a1)+      ; copy string and count its
              beq     stop           ; ..length
              addq.w  #1,d0
              bra     copy_count
stop:         nop
str:     dc.b"hi!",NULL
copy:         ds.b5
```

Pass 1:			Location Counter (Address)	Quick translation
NULL	equ	0	NA	
start:	clr.w	d0	0	XXXX
	lea	str,a0	2	XXXX
			4	$aaaa_h$ address
			6	$aaaa_l$ of str
	lea	copy,a1	8	XXXX
			10 (0A)	$aaaa_h$ address
			12 (0C)	$aaaa_l$ of copy
copy_count :	move.b	(a0)+,(a1)+	14 (0E)	XXXX
	beq	stop	16 (10)	XXXX
			18 (12)	dddd disp = stop – $(LC_{instruction} + 2)$
	addq.w	#1,d0	20 (14)	XXXX
	bra	<u>copy_count</u>	22 (16)	XXXX
				dddd copy_count – $(LC_{instruction} + 2)$
stop:	nop		26 (1A)	XXXX
str:	dc.b	"hi!",NULL	28 (1C)	
copy:	ds.b	5	32 (20)	

At the end of pass 1, the symbol table would be:

SYMBOL:	VALUE:	ADDRESS:
NULL	0 (00)	N
start	0 (00)	Y
copy_count	14 (0E)	Y
stop	26 (1A)	Y
str	28 (1C)	Y
copy	32 (20)	Y

Pass 2 would then generate the following code (absolute addresses and relative offsets are highlighted):

```
0       4240        ☐ clr.w              d0
2       41F9        ☐ lea         str,a0
4       0000
6       001C
8       43F9        ☐ lea         copy,a1
A       0000
C       0020
E       12D8        ☐ move.b      (a0)+,(a1)+
10      6700        ☐ beq         stop
12      0008            dest - (instruction + 2) = 1A - (10 + 2) = 08
14      5250        ☐ addq.w      #1,d0
16      6000        ☐ bra         copy_count
18      FFF6            dest - (instruction + 2) = 0E - 18 = -000A = FFF6
1A      4E71        ☐ nop
1C      6869        ☐ dc.b        "hi!",NULL
1E      2100
20      0000        ☐ ds.b        5
22      0000
24      00
```

1 ½ Pass Assembler

- Forward references make it impossible to do a 1 Pass Assembler.

- However, notice that in pass 1 in order to determine addresses for symbols the translation is virtually done.
- Therefore, it is possible to combine the symbol table generation and translation, with forward references, i.e. symbols not on the symbol table, added to a forward reference list and a space holder put in the machine code. At the end of end of the first pass the forward reference list is resolve by filling in place holders with the appropriate value. Since this involves processing the machine code again it is considered a ½ pass.

Loader

- The assembly process produces an “executable image”, saved to disk as a file.
- The loader is the program which loads the image from disk into RAM. It may also:
    - reserve stack space, and initialize SP
    - jump to the program’s first instruction
- The program isn’t normally loaded at address 0.
- What in the executable is affected by not starting at address 0?
    - Absolute addresses (references)
- How do these need to be adjusted?
    - Need to add the new starting address

Loader

- The executable file produced by the assembler actually consists of:
  - the machine language code
  - a <u>relocation list</u>: a list of all locations which contain absolute addresses
- (note: the symbol table is NOT retained unless the debugging flag is on!)

- Once the code image has been loaded into RAM at address *A*, the offset *A* (the <u>relocation constant</u>) is added to each absolute address listed in the relocation list.

Assuming the code was loaded at address 005AF000, the code would look like:

```
005AF000      4240          □ clr.w      d0
005AF002      41F9          □ lea        str,a0
005AF004      005A
005AF006      F01C
005AF008      43F9          □ lea        copy,a1
005AF00A      005A
005AF00C      F020
005AF00E      12D8          □ move.b     (a0)+,(a1)+
005AF010      6700          □ beq        stop
005AF012      0008              dest – (instruction + 2) = 1A – (10 + 2) = 08
005AF014      5250          □ addq.w     #1,d0
005AF016      6000          □ bra        copy_count
005AF018      FFF6              dest – (instruction + 2) = 0E – 18 = -000A = FFF6
005AF01A      4E71          □ nop
005AF01C      6869          □ dc.b"hi!",NULL
005AF01E      2100
005AF020      0000          □ ds.b5
005AF022      0000
005AF024      00
```

Loader

- The loader is part of the O/S. It is invoked whenever you run a program.
- The O/S (usually) remains in RAM while the program executes. When the program requests termination, the O/S takes back over and unloads the program.

Actual Atari executable program

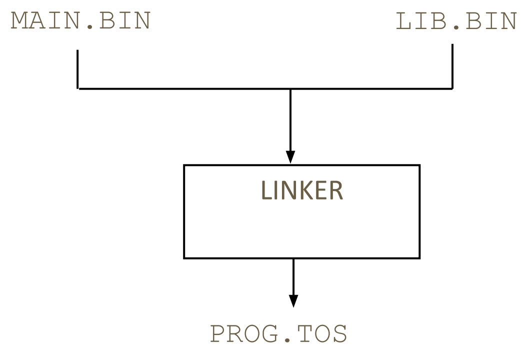
Trans_EX_HEX.pdf created by using a program as input to the C:\BIN\UNHEX2.PRG

TRANS_EX_HEX_DECOMPOSED.pdf

explains what everything means

Linker

- A Linker is a program that combines multiple object modules into an executable module.



Linker

- Each object module contains:
    - machine code
    - a relocation list – absolute addresses that need to be relocated
    - a global reference list - the global symbols it exports to other modules, this is a series of symbol names and locations in the module
    - an external reference list – the global symbols it imports from other modules, this is a series of names and locations in the module that need to be updated with an actual address.
- When linking regardless of the number of modules to be linked the process is done pairwise starting from the front of the list.

Linker

The linking process consists of:

- Combining the machine code of the two modules:
  - the contents of the first module are unchanged
  - the contents of the second module need to be relocated. This involves:
    - updating the second module’s relocation list, both the items in code and the addresses on the relocation list. This is identical to loading except the relocation constant is the size of the first module.
    - updating all addresses on both the global reference list and external reference list need to be updated by adding the relocation constant.

Linker

The linking process consists of (continued):

- Attempting to resolve the first module's external reference list with the second module's global reference list.
- Attempting to resolve the second module's external reference list with the first module's global reference list.
- Any item that is resolved is removed from the external reference list.

Linker

The linking process consists of (continued):

- Combining all of the lists:
  - Relocation lists
  - Global reference lists
  - External reference lists.
- During this process an error is generated if multiple modules both define the same global symbol
- At the end of this process the combined external reference list must be empty; otherwise there are unresolved references, i.e. undefined symbols.

Linker

In Devpac:

- the xref assembler directive simply tells the assembler not to panic when a reference remains unresolved.
- The xdef assembler directive tells the assembler to maintain the symbol table entry for the specified labels.

Assemble each of the two modules below (show the machine language code, the global reference table, the external reference list and relocation list for each). Then, link the two (show the resulting machine language code and relocation list).

MAIN.S

```
            xref        sub1,sub2
 start:     jsr         sub1
            jsr         sub2
            clr.w       -(sp)        ; exit
            trap        #1
```

LIB.S

```
            xdef        sub1,sub2
sub1:       nop
            rts

sub2:       nop
            rts
```

MAIN.BIN (all numbers shown in hex):

```
Global Reference List:   empty
External Reference List:
    sub100000002A
    sub200000008A
Relocation Map List:
    00000002
    00000008
```

Text:	Address:
4EB9	0x00
0000	0x02
0000	0x04
4EB9	0x06
0000	0x08
0000	0x0A
4267	0x0C
4E41	0x0E

LIB.BIN (all numbers shown in hex):

```
Global Reference List:
    sub1 = 000000
    sub2 = 000004
External Reference List:empty
Relocation List:        empty
```

```
Text:           Address:
    4E71    0x00
    4E75    0x02
    4E71    0x04
    4E75    0x06
```

Summary of info in executable file

```
Global Reference List:
    sub1 = 000010
    sub2 = 000014
External Reference List:   empty
Relocation List:
    000002
    000008
```

Text:	Address:
4EB9	0x00
0000	0x02
0010	0x04
4EB9	0x06
0000	0x08
0014	0x0A
4267	0x0C
4E41	0x0E
4E71	0x10
4E75	0x12
4E71	0x14
4E75	0x16

Note: The addresses related to the code in lib.bin have been adjusted