**The Assembler**

* Assembly code (jr $31) → assembler → machine code (01101…)
* Any translation involves 2 phases
  + **Analysis** – understand what is meant by the source string
  + **Synthesis** – output equivalent target string
* **Assembly file** – stream of characters
  + First group the chars into meaningful tokens
    - E.g. label, hex #, .word, etc.
  + Group tokens into instructions (if possible); aka. Analysis
  + If the tokens do not form sensible instructions, output ‘ERROR’ to stderr
    - Focus on checking correct cases rather than error cases
* How to assemble program branches?
  + Assemble in 2 passes
  + **Pass 1 – analysis:**
    - Group tokens into instructions
    - Record addresses of labelled instructions in a “symbol table” – list of (label, address) pairs
      * A line of assembly can have more than one label
      * A word can be labelled after the end of a file
    - This pass checks for syntax errors (e.g. improper form/structure) and semantic errors (e.g. defining the same label twice)
    - Output of pass 1: the symbol table & an intermediate representation
      * **Symbol table** – maps labels to addresses
        + E.g. loop → 0x08; end → 0x2c; etc.
        + Can be implemented with a map in C++
      * **Intermediate representation** – remove comments, create tokens, keeps program as ASCII/Unicode chars
    - Pseudocode:

PC = 0

For each line of input

Scan line

For each LABEL

If already in symbol table

ERROR and exit

Else

Add (LABEL, PC) to symbol table

End for

If next token is an OPCODE

If remaining tokens are not as expected

ERROR and exit

Else

Create intermediate representation of instruction

PC += 4

End for

* + **Pass 2 – synthesis:**
    - Translate each instruction into machine code
    - If an instruction refers to a label, look up the corresponding address in symbol table
    - Output assembled MIPS code to stdout
    - Output symbol table to stderr
    - Pseudocode:

For each OPCODE in the intermediate representation

Translate to MIPS instruction

Look up any labels in symbol table and find address for label

Output instruction (as 4 bytes)

End for

* **Loader** – program responsible for copying a program from secondary storage (hard drive) into primary storage (RAM) and begin its execution
  + Takes program P as input
  + Determines length of P
  + Finds location in RAM for P → α = starting address
  + Allocate RAM starting at α for the code and the program stack
  + Copies P into RAM starting at address α
  + Loads α into a register (e.g. $5)
  + Begins execution at α (jalr $5)
  + E.g.
    - .word P(1) … .word P(k) ; load instructions
    - n = k + space needed for stack
    - a = first address of n contiguous words of RAM
    - for i = 1 … k ; copy instructions into RAM
      * MEM[a + (i – 1)\*4] ← P(i)
    - $30 = a + 4\*n ; set stack pointer
    - jalr $5 ; $3 contains address a
* **Relocation**
  + Label address values need to be adjusted when the program is shifted around in memory
  + .word label – add α to label address
  + .word constant – don’t need to adjust
  + beq, bne – values refer to relative addresses − don’t need to adjust
* Assembled file is a stream of bits – how to differentiate between .word (with ids) and instructions?
  + Need more info from the assembler
  + The output of most assemblers is not pure machine code
  + **Object file** – contains binary machine code & auxiliary info needed by the loader & linker
  + Our object file format – MERL – MIPS executable relocatable linkable
  + Need to contain:
    - Machine code
    - Which lines of code (addresses) were originally .word id
* **MERL format** – 3 parts
  + Header – 3 words
    - Always first word: 0x1000 0002 – cookie – indicates this is a MERL file
      * MIPS for beq $0, $0, 2 – i.e. command to skip the header
      * Thus MERL files can be executed as ordinary MIPS files if loaded at 0
    - Second word: length of MERL file (all 3 parts)
      * i.e. pointer to address right after end of file
    - Third word: code length = length of header + MIPS
      * i.e. pointer to first line of footer
  + MIPS binary
    - Assembled to start at α + 0x0c
  + Symbol table (footer)
    - Format code = 0x1 (indicates relocation)
    - Address of a relocatable word in MIPS section, i.e. address of .word instructions
    - Format code, address, format code, address, …
  + Relocation tool: cs241.merl
    - Input – MERL file & relocation address
    - Output – non-relocatable MIPS file with MERL header & footer removed, ready to load at the given address
  + mips.twoints/array – have optional 2nd argument = address to load the file at
  + E.g. load myobj.merl at 0x1000
    - Java cs241.merl 0x1000 < myobj.merl > myobj.mips
    - Java mips.twoints myobj.mips 0x10000
* Load relocation algorithm

read() // skip cookie; do nothing

endMod ← read() – 12 //length of code + footer (not including header)

codeLen ← read() – 12 //length of code (not including header)

α ← findFreeRAM(codeLen + stack)

for (int i = 0; i < codeLen; i += 4)

MEM[α + i] ← read()

end for

while (i < endMod)

formatCode ← read()

if (formatCode == 1)

rel ← read() //address to be relocated

//adjust forward by α & backwards by header length

MEM[rel + α – 12] += α – 12

else ERROR

i += 8

end while

* **Linkers**
  + Problem: how can the assembler resolve a reference to a label in a different file?
  + Solution 1: cat the files together, then assemble the result
  + Solution 2: a tool that understands MERL files & can combine them intelligently – a **linker**
  + The assembler must be modified to handle references to labels that aren’t there
  + When the assembler encounters .word id where label id is not found, it fills in 0 for id and indicates that the program requires the value of id before it can run
  + Makes an entry in the MERL footer
  + Error-checking ability is lost if the assembler simply assumes any labels that are not found are external references
    - Use new directive: **.import id**
    - Asks the assembler for id to be linked in
    - Recall directives do not compile into a word of MIPS
  + When assembler encounters .word id, if label id is not found and no .import id, then it is an error
  + External symbol reference (ESR)
    - Records where the symbol is being used (where is the 0 that needs to be filled in)
    - Records the name of the symbol
    - Format:
      * Word 1 – 0x11 (format code)
      * Word 2 – location of symbol usage
      * Word 3 – length of name of symbol in chars
      * Word 4 … – ASCII characters in the name (1 char in each word)
  + How to know which label id to link to? (in case of duplication)
    - Another new directive: **.export id**
    - Tells the assembler to make an entry in the MERL symbol table
  + External symbol definition (ESD)
    - Word 1 – 0x05 (format code)
    - Word 2 – address the symbol represents
    - Word 3 – length of name of symbol in chars
    - Word 4 … − ASCII chars in name
* **Linker algorithm**
* Input – MERL files m1 & m2
  + i.e. header 1, MIPS 1, footer 1, header 2, MIPS 2, footer 2
* Output – single MERL file with m2 linked after m1
  + i.e. header, MIPS 1, MIPS 2, footer
* Pseudocode:
  + α = m1.codeLength – 12
  + Must relocate m2.code by α
  + Add every address in m2.symbolTable by α
  + If m1.exports.labels ∩ m2.exports.labels != 0
    - ERROR //both files are exporting the same label
  + For each (address1, label) in m1.imports
    - If ∃ (address2, label) in m2.exports //import-export match
      * m1.code[address1] ← address2 //replace 0 with the right address
      * Remove (address1, label) from m1.imports
      * Add address1 to m1.relocates //address1 is relocatable
  + For each (address2, label) in m2.imports
    - If ∃ (address1, label) in m1.exports //same thing with 1 & 2 switched
      * m2.code[address2] ← address1
      * Remove from (address2, label) from m2.imports
      * Add address2 to m2.relocates
  + Imports = m1.imports ∪ m2.imports
  + Exports = m1.exports ∪ m2.exports
  + Relocates = m1.relocates ∪ m2.relocates
  + Output MERL cookie
  + Output total codeLength + total symbolTable.length + 12
  + Output total codeLength + 12
  + Output m1.code
  + Output m2.code
  + Output imports, exports, relocates