university of groningen

Computer Architecture 2023-24 (WBCS010-05)

Lecture 8: Assembly

Reza Hassanpour
r.zare.hassanpour@rug.nl

Human-Friendly Programming

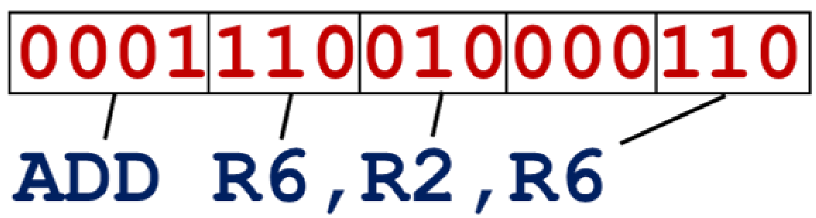
- Computers need binary instruction encodings…
  - `0001110010000110`
- Humans prefer symbolic languages…
  - `a = b + c`
- High-level languages allow us to write programs in clear, precise language that is more like English or math. Requires a program (compiler) to translate from symbolic language to machine instructions.
- Examples: C, Python, Fortran, Java, ...

Assembly Language: Human-Friendly ISA Programming

- Assembly Language is a low-level symbolic language, just a short step above machine instructions
  - Don't have to remember opcodes (ADD = 0001, NOT = 1001, ...)
  - Give symbolic names to memory locations -- don't have to do binary arithmetic to calculate offsets
  - Like machine instructions, allows programmer explicit, instruction-level specification of program
- Disadvantage:
  - Not portable. Every ISA has its own assembly language. Program written for one platform does not run on another.

Assembly Language

- Very similar format to instructions -- replace bit fields with symbols



- For the most part, one line of assembly language = one instruction
- Some additional features for allocating memory, initializing memory locations, service calls
- Numerical values specified in hexadecimal (x30AB) or decimal (#10)

x10 is not the same as #10 !

university of groningen

Example Program

```
;
; Program to multiply a number by the constant 6
;
          .ORIG   x3050
          LD      R1, SIX
          LD      R2, NUMBER
          AND     R3, R3, #0      ; Clear R3.  It will
                                  ; contain the product.
; The inner loop
;
AGAIN     ADD     R3, R3, R2
          ADD     R1, R1, #-1     ; R1 keeps track of
          BRp     AGAIN           ; the iteration.
;
          HALT
;
NUMBER    .BLKW   1
SIX       .FILL   x0006
;
          .END
```

Instructions

Comments

Assembler Directives

Labels

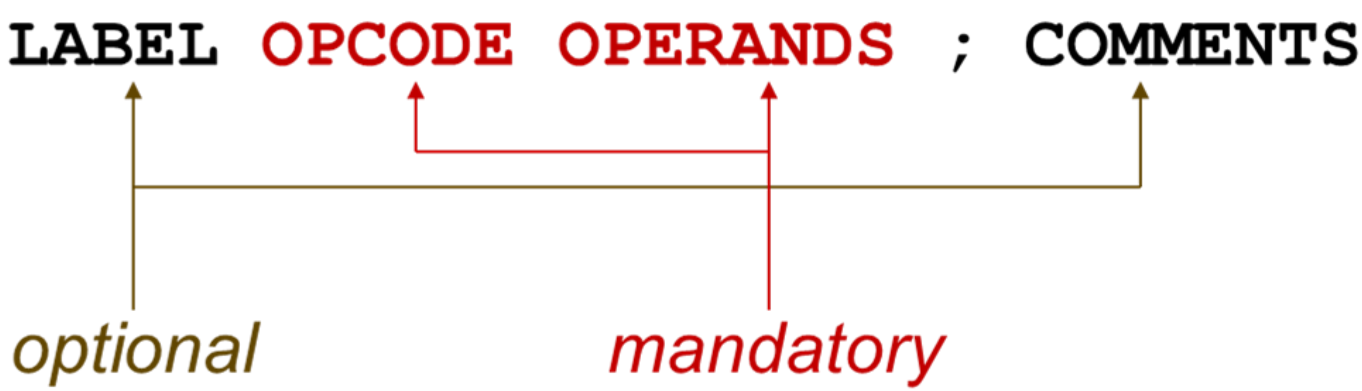
Assembly Language Syntax

› Each line of a program is either one of the following:
- An instruction
- An assembler directive (or pseudo-op)
- A comment

› Whitespace (between symbols) and case are ignored.

› Comments (beginning with “;”) are also ignored.

› An instruction has the following format:



Mandatory: Opcode and Operands

› Opcodes

Reserved symbols that correspond to LC-3 instructions.

Listed in Appendix A and Figure 5.3.

- For example: `ADD`, `AND`, `LD`, `LDR`, ...

› ***reserved*** means that it cannot be used as a label

› Operands

- Registers -- specified by Rn, where n is the register number.
- Numbers -- indicated by # (decimal) or x (hex).
- Label -- symbolic name of memory location (1 to 20 alphanumeric characters)
- Separated by comma (whitespace ignored).
- Number, order, and type correspond to instruction format.

```
ADD     R1,R1,R3    ; DR, SR1, SR2
ADD     R1,R1,#3    ; DR, SR1, Imm5
LD      R6,NUMBER   ; DR, address (converted to PCoffset)
BRz     LOOP        ; nzp becomes part of opcode, address
```

Optional: Label and Comment

› Label

- Placed at the beginning of the line
- Assigns a symbolic name to the address corresponding to that line

```
LOOP ADD   R1,R1,#-1   ; LOOP is address of ADD
     BRp   LOOP
```

› Comment

› A semicolon, and anything after it on the same line, is a comment

› Ignored by assembler

› Used by humans to document/understand programs

› Tips for useful comments:

- Avoid restating the obvious, as “decrement R1”
- Provide additional insight, as in “accumulate product in R6”
- Use comments and empty lines to separate pieces of program

Assembler Directive

› Pseudo-operation

- Does not refer to an actual instruction to be executed
- Tells the assembler to do something
- Looks like an instruction, except "opcode" starts with a dot

Opcode	*Operand*	*Meaning*
`.ORIG`	**address**	**starting address of program**
`.END`		**end of program**
`.BLKW`	**n**	**allocate n words of storage**
`.FILL`	**n**	**allocate one word, initialize with value n**
`.STRINGZ`	**n-character string**	**allocate n+1 locations, initialize with characters and null terminator**

.ORIG

- .ORIG tells the assembler where in memory to place the LC-3 program.
- Example: .ORIG x3050 says, place the first LC-3 ISA instruction in location x3050.

- If the program consists of x100 LC-3 instructions, and .ORIG says to put the first instruction in x3050, the remaining xFF instructions are placed in locations x3051 to x314F.

university of groningen

.END

- .END tells the assembler it has reached the end of the program and need not even look at anything after it.
- Any characters that come after .END will not be processed by the assembler.
- .END does not stop the program during execution.
- In fact, .END does not even exist at the time of execution.

.FILL

- .FILL tells the assembler to set aside the next location in the program and initialize it with the value of the operand.
- The value can be either a number or a label.
- Example
  - TEN .FILL #10
- Example
  - .ORIG x3000
  - AND R1, #0
  - LOOP ADD R1, R1, #1
  - …….
  - FIRST .FILL LOOP

.BLKW

- .BLKW tells the assembler to set aside some number of sequential memory locations (i.e., a BLocK of Words) in the program.
- The actual number is the operand of the .BLKW pseudo-op.
- Example
- MyArray .BLKW #5

.STRINGZ

- .STRINGZ tells the assembler to initialize a sequence of n+1 memory locations.
- The argument is a sequence of n characters inside double quotation marks.
- The first n words of memory are initialized with the zero-extended ASCII codes of the corresponding characters in the string.
- Example:

.STRINGZ

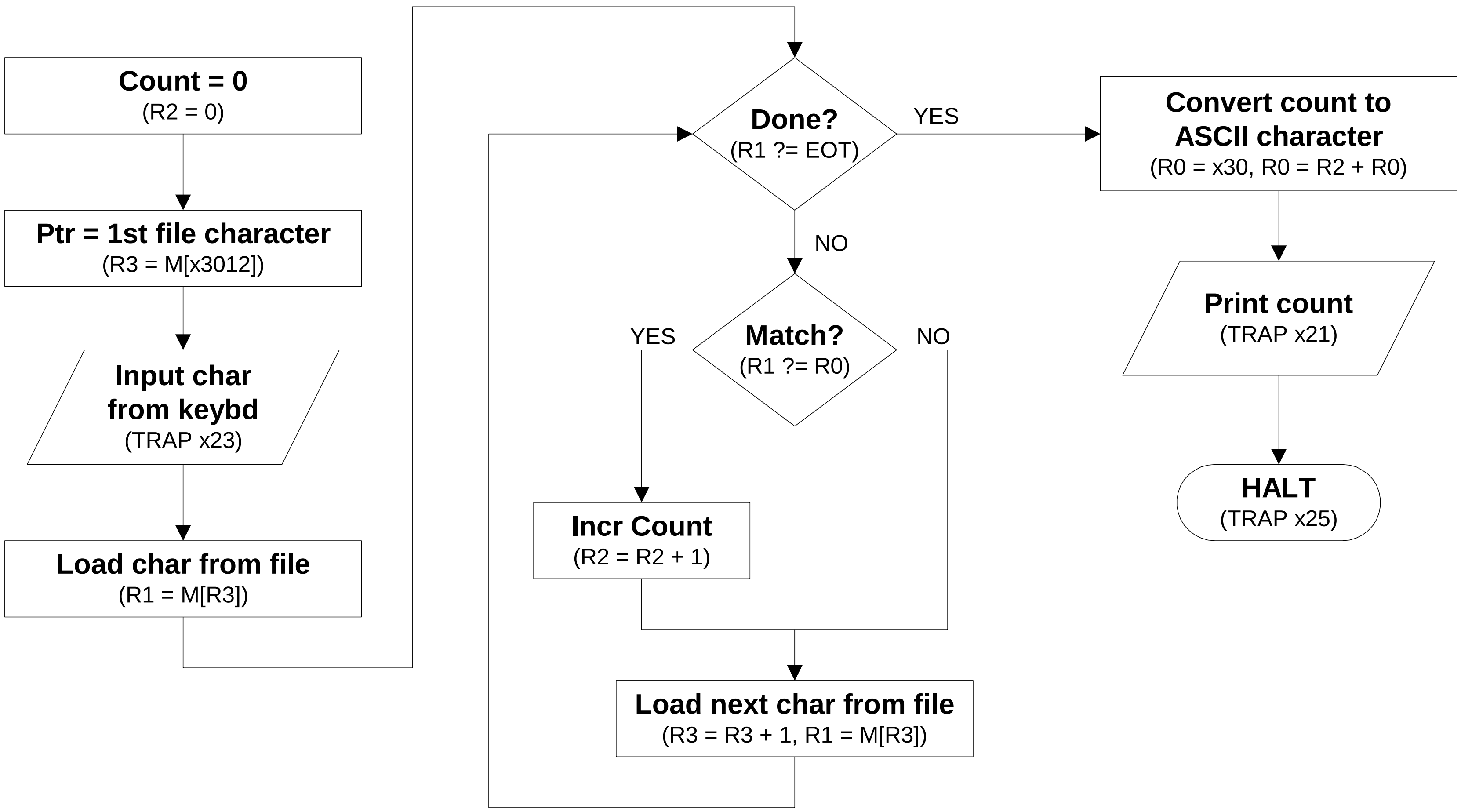
- For example, the code fragment
- .ORIG x3010
- HELLO .STRINGZ "Hello, World!"

- would result in the assembler initializing locations x3010 through x301D to the following values:

- x3010: x0048
- x3011: x0065
- x3012: x006C
- x3013: x006C
- x3014: x006F
- x3015: x002C
- x3016: x0020
- x3017: x0057
- x3018: x006F
- x3019: x0072
- x301A: x006C
- x301B: x0064
- x301C: x0021
- x301D: x0000

Sample Program: Counting Occurrences in a File

› Once again, we show the program that counts the number of times (up to nine) a user-specified character appears in a file.



Assembly Language Program 1

```
;
; Program to count occurrences of a character in a file.
; Character to be input from the keyboard.
; Result to be displayed on the monitor.
; Program only works if no more than 9 occurrences are found.
;
;
; Initialization
;
        .ORIG     x3000
        AND       R2, R2, #0           ; R2 is counter, initially 0
        LD        R3, PTR              ; R3 is pointer to characters
        TRAP      x23                  ; R0 gets character input
        LDR       R1, R3, #0           ; R1 gets first character
;
; Test character for end of file
;
TEST    ADD       R4, R1, #-4          ; Test for EOT (ASCII x04)
        BRz       OUTPUT               ; If done, prepare the output
;
; Test character for match.  If a match, increment count.
;
        NOT       R1, R1
        ADD       R1, R1, #1
        ADD       R1, R1, R0       ; Compute R0-R1 to compare
        BRnp      GETCHAR          ; If no match, do not increment count
        ADD       R2, R2, #1
```

Assembly Language Program 2

```
;
; Get next character from file.
;
GETCHAR         ADD R3, R3, #1   ; Point to next character.
        LDR     R1, R3, #0   ; R1 gets next char to test
        BRnzp   TEST
;
; Output the count.
;
OUTPUT  LD      R0, ASCII    ; Load the ASCII template
        ADD     R0, R0, R2   ; Covert binary count to ASCII
        TRAP    x21          ; ASCII code in R0 is displayed.
        TRAP    x25          ; Halt machine

;
; Storage for pointer and ASCII template
;
ASCII   .FILL   x0030
PTR     .FILL   x4000
        .END
```

- What if we don’t put HALT (TRAP x25) at the end of the program?

Data or Instruction?

```
› OUTPUT LD      R0, ASCII    ; Load the ASCII template
›        ADD     R0, R0, R2   ; Covert binary count to ASCII
›        TRAP    x21          ; ASCII code in R0 is displayed.

› ;
› ; Storage for pointer and ASCII template
› ;
› ASCII  .FILL   x0030
› PTR    .FILL   x4000
›        .END
```

- Next memory location after TRAP x21 contains x0030
- In binary: 0000 000 000110000
- Branch to PC + 48 if ?
- x4000 = 0100 000 000 000000 (Jump to subroutine)

Assembly Language Program 3

```
        .ORIG x3000
        AND R5, R5, #0
        AND R3, R3, #0
        ADD R3, R3, #8
        LDI R1, A
        ADD R2, R1, #0
AG      ADD R2, R2, R2
        ADD R3, R3, #-1
        BRnp AG
        LD R4, B
        AND R1, R1, R4
        NOT R1, R1
        ADD R1, R1, #1
        ADD R2, R2, R1
        BRnp NO
        ADD R5, R5, #1
NO      HALT
B       .FILL xFF00
A       .FILL x4000
        .END
```

Assembly Language Program 4 (I)

```
        .ORIG x3000
ONE     LD R0, A
        ADD R1, R1, R0
TWO     LD R0, B
        ADD R1, R1, R0
THREE   LD R0, C
        ADD R1, R1, R0
        ST R1, SUM
        TRAP x25
A       .FILL x0001
B       .FILL x0002
C       .FILL x0003
SUM     .FILL x0004
        .END
```

Assembly Language Program 4 (II)

```
      .ORIG x3000
      AND R1, R1, #0
ONE   LD R0, A
      ADD R1, R1, R0
TWO   LD R0, B
      ADD R1, R1, R0
THREE LD R0, C
      ADD R1, R1, R0
      ST R1, SUM
      TRAP x25
A     .FILL x0001
B     .FILL x0002
C     .FILL x0003
SUM   .FILL x0004
      .END
```

```
.ORIG x3000
      AND R1, R1, #0
ONE   LD R0, A
      ADD R1, R1, R0
TWO   LD R0, B
      ADD R1, R1, R0
THREE LD R0, C
      ADD R1, R1, R0
      LD R0, ONE
      LDI R0, ONE
      ST R1, SUM
      TRAP x25
A     .FILL x0001
B     .FILL x0002
C     .FILL x0003
SUM   .FILL x0004
      .END
```

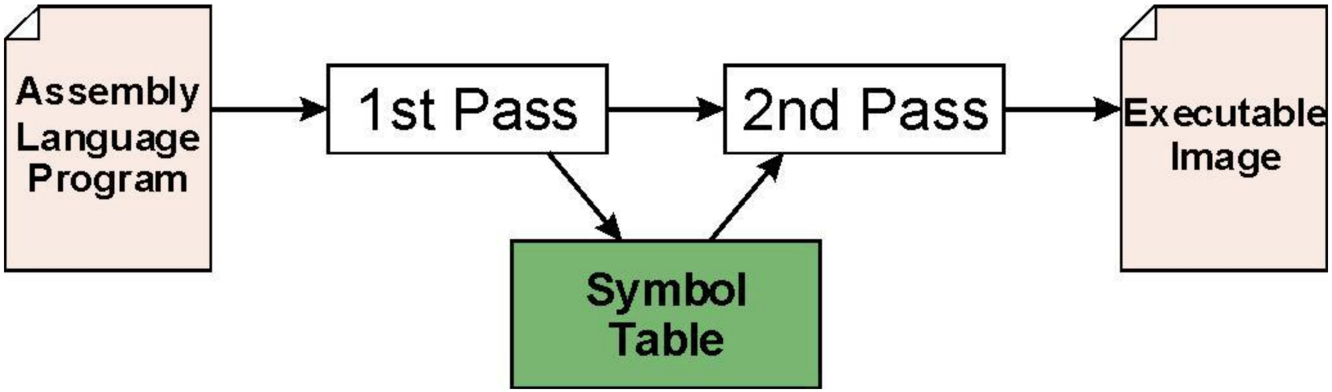
Assembly Language Program 4 (III)

- LD R0, ONE   will load the value of the location shown by label ONE into R0. In this example, ONE is x3001.
- Content of x3001 is LD R0, A = 0010 000 000001001
- R0 will contain x2009

- LDI R0, ONE will load the content of the location shown by an address stored at x3001.
- This is equivalent to R0 ← M[x2009]

Assembly Process

› The assembler is a program that translate an assembly language (.asm) file to a binary object (.obj) file that can be loaded into memory.



› **First Pass:**

- Scan program file, check for syntax errors
- Find all labels and calculate the corresponding addresses: the *symbol table*

› **Second Pass:**

- Convert instructions to machine language, using information from symbol table

First Pass: Construct the Symbol Table

1. Find the .ORIG statement,
   which tells us the address of the first instruction
   - Initialize location counter (LC), which keeps track of the current instruction
2. For each non-empty line in the program:
   - If line contains a label, add label and LC to symbol table
   - Increment LC
   - NOTE: If statement is .BLKW or .STRINGZ, increment LC by the number of words allocated
3. Stop when .END statement is reached

› **NOTE:** A line that contains only a comment is considered an empty line

First Pass on Sample Program (Comments Removed)

```
› --              .ORIG   x3000
› x3000           AND     R2, R2, #0
› x3001           LD      R3, PTR
› x3002           TRAP    x23
› x3003           LDR     R1, R3, #0
› x3004  TEST     ADD     R4, R1, #-4
› x3005           BRz     OUTPUT
› x3006           NOT     R1, R1
› x3007           ADD     R1, R1, #1
› x3008           ADD     R1, R1, R0
› x3009           BRnp    GETCHAR
› x300A           ADD     R2, R2, #1
› x300B  GETCHAR  ADD     R3, R3, #1
› x300C           LDR     R1, R3, #0
› x300D           BRnzp   TEST
› x300E  OUTPUT   LD      R0, ASCII
› x300F           ADD     R0, R0, R2
› x3010           TRAP    x21
› x3011           TRAP    x25
› x3012  ASCII    .FILL   x0030
› x3013  PTR      .FILL   x4000
› --              .END
```

Label	Address
TEST	x3004
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

Second Pass: Convert to Machine Instructions

1. Find the .ORIG statement,
   which tells us the address of the first instruction.
   - Initialize location counter (LC), which keeps track of the current instruction
2. For each non-empty line in the program:
   - If line contains an instruction, translate opcode and operands to binary machine instruction. For label, lookup address in symbol table, subtract (LC+1) and replace label with that. Increment LC
   - If line contains .FILL, convert value/label to binary. Increment LC
   - If line contains .BLKW, create n copies of x0000 (or any arbitrary value). Increment LC by n
   - If line contains .STRINGZ, convert each ASCII character to 16-bit binary value. Add null (x0000). Increment LC by n+1
3. Stop when .END statement is reached

Example

- .ORIG x3000
- AND R2,R2,#0 ; R2 is counter, initialize to 0
- LD R3,PTR ; R3 is pointer to characters

Symbol	Address
TEST	x3004
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

- Set LC to x3000
- AND R2,R2,#0 ➔ 0101010010100000
- Increment LC ➔ LC = x3001
- LD R3,PTR ➔ 0010011000010001
- PTR is x3013 from Symbol table
- Subtract LC+1 from x3013 ➔ x3013 – x3002 ➔ x0011
- X0011 ➔ 000010001 (9 bits binary)
- Increment LC ➔ LC = x3002

Errors during Code Translation

› While assembly language is being translated to machine instructions, several types of errors may be discovered

- Immediate value too large -- can't fit in Imm5 field
- Address out of range -- greater than LC+1+255 or less than LC+1-256
- Symbol not defined, not found in symbol table

› If error is detected, assembly process is stopped and an error message is printed for the user

Beyond a Single Object File

- Larger programs may be written by multiple programmers, or may use modules written by a third party. Each module is assembled independently, each creating its own object file and symbol table.
- To execute, a program must have all of its modules combined into a single executable image
- **Linking** is the process to combine all of the necessary object files into a single executable

External Symbols

- In the assembly code we're writing, we may want to symbolically refer to information defined in a different module
- For example, suppose we don't know the starting address of the file in our counting program. The starting address and the file data could be defined in a different module.
- We want to do this:
  - `PTR .FILL  STARTofFILE`
- To tell the assembler that `STARTofFILE` will be defined in a different module, we could do something like this:
  - `.EXTERNAL STARTofFILE`
- This tells the assembler that it's not an error that `STARTofFILE` is not defined. It will be up to the linker to find the symbol in a different module and fill in the information when creating the executable.

Questions?