COMP 2280 - Introduction to Computer Systems

Module 3- Introduction to Instruction Set Architecture (ISA) and Assembly Language

ISA - Introduction

- The instruction set architecture (ISA) provides the interface between computer programs and the hardware.
- The *ISA* is the part of the processor (CPU) that is visible to the programmer.
- Each type of CPU architecture has its own ISA.
- The instruction set of the ISA specifies the commands that can be used to program a CPU or the operations that can be performed by the CPU.

ISA - Introduction

- You must distinguish between ISA and how it is implemented.
- Example – Consider the IA32 (3rd gen of x86 ISA, i386)
  - Intel and AMD have vastly different implementations for the same ISA.
  - Programs written for one will run on the other.
- There are other ISAs, which are not binary compatible with IA32
  - PowerPC (Pre-2005 Macs & Consoles)
  - MIPs
  - Sparc
  - ARM (Mobile devices)
  - M68000 (Motorola 68k)

ISA - Introduction

- ISA = All of the *programmer-visible* components and operations of the computer
  - memory organization
    - address space -- how may locations can be addressed?
      - "How many boxes do I have?"
    - addressability -- how many bits per location?
      - "How big are my boxes?"
  - register set
    - how many? what size? how are they used?
  - instruction set
    - Opcodes (ADD/0001, AND/0101, LEA/1110)
      - Book: The **Opcode** (symbolic name) matches the **opcode** (bits) of the instruction
    - Data types (floating point, integers, etc.)
    - Addressing modes
      - Direct: "Go to place X"
      - Indirect: "Go to the X^{th} address of the phonebook"
      - Offset: "Go X steps to the left from your current location"
- ISA provides all information needed for someone that wants to write a program in machine language
  (or translate from a high-level language to machine language).

ISA - Introduction

- Instructions can be classified as:
  - Data Processing:
    - Arithmetic and Logic instructions
  - Data Storage:
    - Memory instructions
  - Control:
    - Comparison/Test and Branch instructions
  - Data Movement:
    - I/O instructions (not always present – memory mapped I/O)

ISA-Design

- The number and types of instructions available in the instruction set dictates its complexity:
  - CISC (complex instruction set computer): provides a large collection of instructions, with some support for HLL (high level language). (x86/x64 is CISC-ish)
  - RISC (reduced instruction set computer): provides fewer instructions, and usually no support for HLL. Idea behind RISC is:
    - Make the common case fast
    - 80:20 rule (use freed space for efficient implementations)
    - ARM is RISC
      - Previously **Advanced RISC Machine**, originally **Acorn RISC Machine**

ISA Design

- There are tradeoffs between a small and large instruction set.
  - Large instruction sets requires more logic to implement, but is easier to use.
  - Small instruction sets are easier to implement, but harder to use.

ISA Design

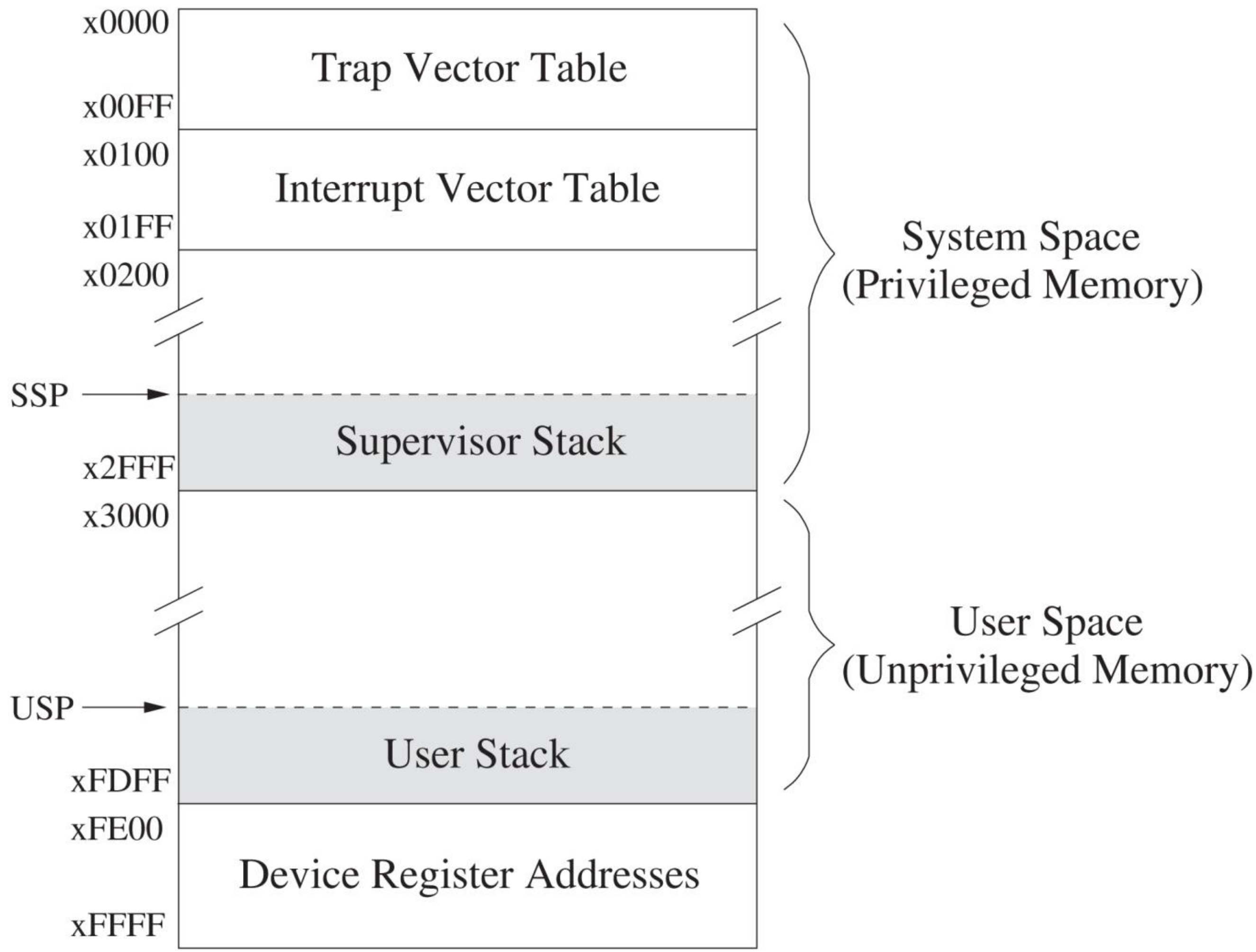
- Examples of RISC Architectures
  - Sparc, MIPs, IBM RS/6000
- Examples of CISC Architectures
  - Motorola 68000, Intel 80386

LC3 ISA

Note: LC3 Instruction set reference: pages 655-674, 3rd edition

- Let us take a look at the LC3 ISA.
  - *Appendix A* of the textbook.
- Memory
  - address space: 2^{16} locations (16-bit addresses)
  - addressability: 16 bits ie. Memory is addressed 16 bits at a time.
  - Here is a layout of LC-3's memory (memory map)

LC3 ISA



Memory Map of the LC-3

LC3 ISA

- Registers
  - *temporary high-speed storage, accessed in a single machine cycle*
    - *accessing regular memory (RAM) generally takes longer than a single cycle*
    - "Sort of / kind of" *like a variable but is not a variable*
  - eight general-purpose registers: R0 - R7
    - each 16 bits wide
  - other registers
    - not directly addressable, but used by (and affected by) instructions
    - PC (program counter)
    - IR (instruction register)
    - condition codes (N = Negative, Z = Zero, P = Positive)

LC3 ISA

- Opcodes (actual instructions available)
  - 15 opcodes
    - Each instruction is 16-bits long, with 4 bit opcode
    - 12 bits for operands and addressing mode
  - *Operate* instructions: ADD, AND, NOT
  - *Data movement* instructions: LD, LDI, LDR, LEA, ST, STR, STI
  - *Control* instructions: BR, JSR/JSRR, JMP/RET, RTI, TRAP
    - some opcodes set/clear *condition codes*, based on result:
      - N = negative, Z = zero, P = positive (> 0)

LC3 ISA

- Data Types
  - 16-bit 2's complement integer
- Addressing Modes
  - How is the location of an operand specified?
  - non-memory addresses: *immediate*, *register*
  - *memory addresses: PC-relative, indirect, base+offset*

LC3 ISA

- Only three arithmetic/logic instructions: ADD, AND, NOT
  - Source and destination operands are registers
    - These instructions *do not* reference memory.
    - ADD and AND can use "immediate" mode, where one operand is hard-wired into the instruction.

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 translage from symbolic language to machine instructions.

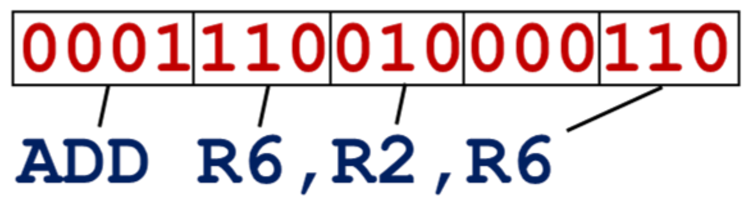
ISA Programming

- 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 hexdecimal (x30AB) or decimal (#10).

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

Note: LC3 Instruction set reference: pages 655-674, 3rd edition

- **LC-3 Assembly Language Syntax**
  - Each line of a program is one of the following:
    - an instruction
    - an assembler directive (or pseudo-op)
    - a comment
  - White space (between symbols) and case is ignored.
  - Comments (beginning with ";") are also ignored.
    - An instruction has the following format:
    - LABEL **OPCODE OPERANDS** ; COMMENTS
      - Bold indicates required, others are optional

Assembly Language

- **Mnemonics**
  - reserved symbols that correspond to LC-3 instructions
  - listed in Appendix A
    - ex: ADD, AND, LD, LDR, …
- **Operands**
  - registers -- specified by Rn, where n is the register number
  - numbers -- indicated by # (decimal) or x (hex)
  - label -- symbolic name of memory location
  - separated by comma
  - number, order, and type correspond to instruction format
  - ex:

```
ADD     R1,R1,R3
ADD     R1,R1,#3
LD      R6,NUMBER
BRz     LOOP
```

Assembly Language

- **Label**
  - placed at the beginning of the line
  - assigns a symbolic name to the address of the corresponding line
  - "Sort of / kind of" like a pointer, but is not a pointer
    - Ex: "*Alex's House*" is the label for the address "*22 Something St*"
  - ex:

```
LOOP    ADD     R1,R1,#-1
        BRp     LOOP
```

- **Comment**
  - anything after a semicolon is a comment
  - ignored by assembler
  - used by humans to document/understand programs
  - tips for useful comments:
    - avoid restating the obvious, as "decrement R1"
      - What **is** R1?
    - provide additional insight, as in "accumulate product in R6"
    - use comments to separate pieces of program
    - Use comments to make the graders lives easier
      - Happy grader, happy grading, happy gradee

Assembler Directives

- **Pseudo-operations (7.2.2 p.236)**
  - do not refer to operations executed by program
  - used by assembler
  - look like instructions, but "mnemonic" starts with a dot (.)
  - Eg)

```
data        .fill #1000 ;1000 dec
sum         .fill x500  ;500 hex
```

Assembler Directives

Directive	*Operand*	*Meaning*
.ORIG	address	starting address of program
.END		end of program
.BLKW	n	allocate n blank words (aka: memory addresses) of storage
.FILL	n	allocate one word, initialize with value n
.STRINGZ	n-character string	allocate n+1 locations, initialize w/characters and null terminator

Traps

- LC-3 assembler provides "pseudo-instructions" for each trap code, so you don't have to remember them.
- A trap provides a mechanism for user (unprivileged) code to execute OS (privileged) code.
- "Pretty much" a system call
- Useful in any system that has an OS.
- For us, traps provide a way to do I/O and halt the machine.

Traps

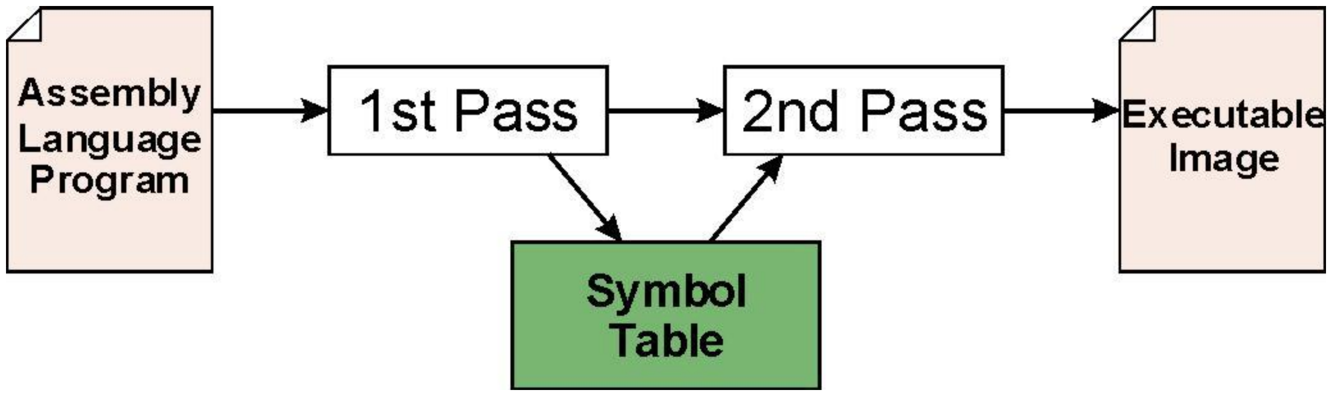
Name	*Equivalent*	*Description*
HALT	TRAP x25	Halt execution and print message to console.
IN	TRAP x23	Print prompt on console, read (and echo) one character from keybd. Character stored in R0[7:0].
OUT	TRAP x21	Write one character (in R0[7:0]) to console.
GETC	TRAP x20	Read one character from keyboard. Character stored in R0[7:0].
PUTS	TRAP x22	Write null-terminated string to console. Address of string is in R0.

Assembly Process

- **The Assembly Process (7.3)**
  - Objective
    - Translate the AL (Assembly Language) program into ML (Machine Language).
    - Each AL instruction yields one ML instruction word.
    - Interpret pseudo-ops correctly.
  - Problem
    - An instruction may reference a label.
    - If the label hasn’t been defined yet, the assembler can't form the instruction word
  - Solution
    - Two-pass assembly

Assembly Process

- Convert assembly language file (.asm) into an executable file (.obj) for the LC-3 simulator.



- First Pass:
  - scan program file
  - find all labels and calculate the corresponding addresses; this is called the *symbol table*
- Second Pass:
  - convert instructions to machine language, using information from symbol table

Assembly Process

- **First Pass - generating 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 location of the current instruction.
- 2. For each non-empty line in the program:
  - a) If line defines a label, add label and LC to symbol table.
  - b) Increment LC.
    - NOTE: If statement is .BLKW or .STRINGZ, increment LC by the number of words allocated.
- 3. Stop when .END statement is reached.

Assembly Process

```
.ORIG    x3050
         LD  R1, SIX
         LD  R2, NUMBER
         AND R3, R3, #0
;
; The inner loop
;
AGAIN
         ADD R3, R3, R2
         ADD R1, R1, b-01
         BRp AGAIN
         ;
         HALT

NUMBER   .BLKW #2
         .STRINGZ "Orphan Text"
SIX      .FILL    x0006

.end
```

Symbol	Address
Again	
Number	
Six	

Obscure sidenote:

- Along with '#' for decimal and 'x' for hexadecimal, binary numbers can also be written in LC3 using 'b'
- This course does not use it in any way, shape, or form other than this example.
- Unless specified otherwise, please use '#' or 'x' and ***do not use 'b'*** in your assignments or labs

Assembly Process

```
;
; Program to multiply a number by  six
;
            .ORIG       x3050
x3050                   LD          R1, SIX
x3051                   LD          R2, NUMBER
x3052                   AND         R3, R3, #0
;
; The inner loop
;
x3053       AGAIN       ADD         R3, R3, R2
x3054                   ADD         R1, R1, b-01
x3055                   BRp         AGAIN
;
x3056                   HALT
;
x3057       NUMBER      .BLKW       #2
X3058                   .STRINGZ "Orphan Text"
x3064       SIX         .FILL       x0006
;
            .END
```

Symbol	Address
Again	x3053
Number	x3057
Six	x3065

Sidenote:

- We have no way to (directly) reference this string…
- Can you think of a work-around without adding a label?

Assembly Process

Memory

Address	Hex	Decimal	Instruction
x3050	x2214	8724	LD R1, SIX
x3051	x2405	9221	LD R2, NUMBER
x3052	x56E0	22240	AND R3, R3, #0
x3053	x16C2	5826	ADD R3, R3, R2
x3054	x127F	4735	ADD R1, R1, b-01
x3055	x03FD	1021	BRp AGAIN
x3056	xF025	61477	HALT
x3057	x0000	0	NUMBER .BLKW #2
x3058	x0000	0	NUMBER .BLKW #2
x3059	x004F	79	O
x305A	x0072	114	r
x305B	x0070	112	p
x305C	x0068	104	h
x305D	x0061	97	a
x305E	x006E	110	n
x305F	x0020	32	
x3060	x0054	84	T
x3061	x0065	101	e
x3062	x0078	120	x
x3063	x0074	116	t
x3064	x0000	0	.STRINGZ "Orphan Text"
x3065	x0006	6	SIX .FILL x0006

Sidenote:

- LC3 shows the memory address, the value at that address (hex and decimal) and the associated instruction line if there is one

Symbol	Address
Again	x3053
Number	x3057
Six	x3065

Assembled code:

- Note the instruction layout (opcode, operands and addressing modes)

Sidenote:

- Like in C, strings end in a null character (x0)
- LC3 places the instruction line there because it's the only spot that makes sense since all strings have a null character

Assembly Process

- **Second Pass - generating the ML program**
  - Scan each line again
  - Translate each AL instruction into ML
    - *Look up symbols in the symbol table*
    - *Ensure that labels are no more than +256 / -255 words from instruction*
    - *Determine operand field for the instruction*
  - Fill memory locations as directed by pseudo-ops
  - Stop when .END is encountered
  - Potential problems:
    - Improper number or type of arguments
      - ex:
        ```
        NOT    R1,#7
        ADD    R1,R2
        ADD    R3,R3,NUMBER
        ```
    - Immediate argument too large
      - ex:
        ```
        ADD    R1,R2,#1023
        ```
    - Address (associated with label) more than +256/-255 words from instruction
      - can't use PC-relative addressing mode

Object File Format

- LC-3 object file contains
  - Starting address (location where program must be loaded), followed by…
  - Machine instructions

```
0011000000000000      .ORIG x3000
0101010010100000      AND R2, R2, #0
0010011000010001      LD R3, PTR
1111000000100011      TRAP x23
.
.
.
```

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

Note: LC3 Instruction set reference: pages 655-674, 3rd edition

- 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.