ENS1161 Computer Fundamentals

Module 5
Applications, programming languages and translation

Module Objectives

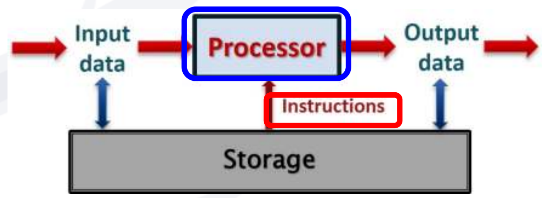
On completion of this module, students should be able to:

- Explain the differences between high-level languages (HLLs) and assembly language, and the advantages and disadvantages of writing programs in each of these.
- Describe the process of translating programs into executable binary code, including the translator tools used.
- Explain how HLL program control structures are implemented using conditional branch instructions.
- Explain what a stack is and how it functions.
- Explain how subroutine calls and returns work, including the role of the stack in these processes.

Moving forward..

- Last module:
  - How the ALU works
  - Flags (condition codes)
  - Arithmetic and Logic instructions
  - Some instructions that use condition codes

- Focus of this module: Programs
  - Programming languages
  - Programming 'tools' that convert programs to binary code



Introduction

- **Module Scope**
  - Levels of programming languages
  - Control structures in programming
  - Tools to convert program code to machine language
    - Roles, advantages and limitations

Using computing devices

- There are a wide variety of computing devices
  - From embedded systems in appliances, to mobile phones and laptops to desktop computers and servers.

- We use these devices though a wide variety of **applications** (*apps*)

Types of software *(recap Module 1)*

- 2 broad categories:
  - **Operating system**
    - Main function is to control the hardware and enable other software to interface with the hardware
    - Also acts as ‘control program’ for other applications
      - e.g. Windows, macOS, Android
  - **Application software**
    - Designed to perform a certain type of function
      - E.g. wordprocessor, browser, spreadsheet, etc.
  - *Operating systems covered in more detail in Module 9*

What is a program? *(recap Module 2)*

- A program is a series of instructions
  - stored in the memory sub-system
  - fetched and executed sequentially
- Each instruction consists of two parts
  - an *operation*
  - an *operand* or *operand address*
    - data / location of the data on which to perform the operation

Operand address

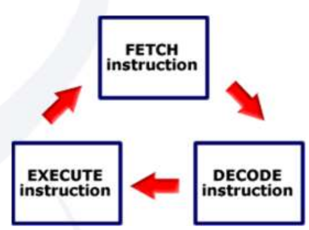
Operations

```
LDA  C100h    ; Load accumulator with data from memory location
              ; C100 hex (or 1100 0001 0000 0000 in binary)
ADDI 2        ; Add immediate value (2) to the number in the
              ; accumulator
```

Operand

How a Computer System Works *(recap)*

- A processor needs a set of instructions
  - tells it what operations to perform on what data
- Instructions (programs) are stored in memory
- The microprocessor:
  - *fetches* an instruction from memory
  - *decodes* it, and
  - *executes* the specified operation
- Sequence of *fetch*, *decode* and *execute* continues indefinitely
  - Until reach an instruction to stop or powered off



Instruction Set *(recap Module 2)*

- A processor can ONLY execute the set of instructions it was built to understand
  - *The set of operations a processor can perform*
  - Different for every processor family
- Each instruction (or variation of it) has a unique opcode
  - Also called object code or machine code
- Each instruction requires specific series of simple steps to complete the required operation
  - Normally done by a built-in *microprogram* (or *microcode*) in the Control Unit
    - Generates the controls signals for the instruction in the right order and at the right time
  - In the *Execution* phase, each opcode results in a separate microprogram being run

Instruction set and programs

- Instruction: binary code interpreted by the CPU to perform an action
- A program is a series of instructions in binary that the processor can read and interpret
- Difficult for humans to write programs in 1s and 0s, so programs are written in more 'human-like' programming languages
  - E.g. assembler, C, Java, Python
- These programs are translated to binary using other programs
  - Assemblers, compilers

Assembly language *(recap Module 2)*

- A program is a series of instructions in binary that the processor can read and interpret (*decode*)
- Difficult to write programs in *1*s and *0*s, so use *assembly language*
- **Assembly language**
  - programs use 3 or 4 character mnemonic English abbreviations
  - mnemonics correspond to the binary opcodes that the processor understands
    - E.g. **LDA** = 1011 0110 (*B6* in hexadecimal), **ADDI** = 1011 1010 (*BA*h)
  - translated to binary using an assembler program

```
LDA  C100h   ; Load accumulator with data from memory location        1011 0110
             ; C100 hex (or 1100 0001 0000 0000 in binary)      →     1100 0001
ADDI 2       ; Add immediate value (2) to the number in the           0000 0000
             ; accumulator                                            1011 1010
                                                                      0000 0010
```


Assembler

- An assembler is a program that converts assembly language programs to object code (binary code / machine language)
  - The process of conversion is called *assembly* or *assembling*
- Also allows programmers to use symbolic names
  - for data, functions and locations within programs
  - The assembler will work out (*resolve*) the actual memory addresses during assembly

```
MOV AL, first_val     ; 1st value into register AL   →   MOV AL, 100h    →   0100 1010
ADD AL, second_val    ; add the 2nd value to AL           ADD AL, 101h        0000 0001
MOV result, AL        ; store the answer in result        MOV 130h, AL        0000 0000
                                                                              0000 0010
                                                                              0000 0001
                                                                              0000 0001
                                                                              0100 1000
                                                                              ……
```

Addresses resolved by assembler

Advantages of programming in assembly

- There are some advantages of programming in assembly language over machine code:
  - Programmers can work with symbolic names
    - As opposed to just binary code or numeric addresses
  - Assemblers can detect certain errors
    - Syntax errors, invalid addressing modes, etc.
  - Programs can be optimised
    - Can be tailored to take full advantage of hardware features
    - However, requires understanding of processor architecture and operation
  - If done properly, code can be more compact

```
MOV AL, first_val
ADD AL, second_val
MOV result, AL
```


Limitations of assembly language

- Requires knowledge of the particular processor's instruction set
  - One-to-one between processor instruction and assembly code
- Requires understanding of the processor's architecture
  - Registers available, addressing modes
- The programs are not *portable*
  - Programs are tied to a particular processor or family of processors
  - Can't be run on other types of processors
- Quite cumbersome
  - Code / steps linked to the way the particular processor works
  - Does not reflect the way human's think or describe their processes
- Solution: High-level languages

```
MOV AL, first_val      ; 1st value into register AL
ADD AL, second_val     ; add the 2nd value to AL
MOV result, AL         ; store the answer in result
```

High-level languages

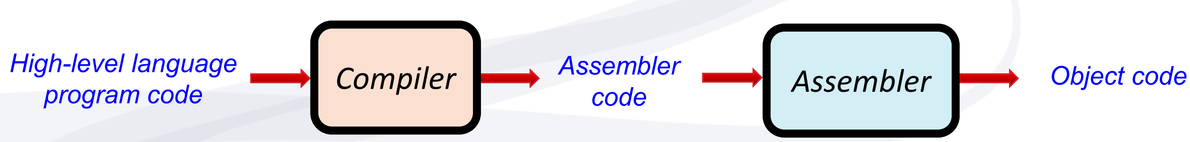
- Programming languages that are designed for human programmers
- Problem-oriented
  - Focuses on program logic

result := first_val + second_val ;

- Abstracts away the machine level details
  - Programmer does not need to know the underlying processor hardware architecture / components
- Makes code portable
  - Not tied to a particular processor / instruction set / architecture
- *Examples of HLL: C, Java, Python*

Compiler

- High-level programs will still need to be translated down to machine code before it can be executed on a processor
- Process is carried out by an application called a *compiler*
  - Compiler processes HLL program code and generates assembler code for the target processor
  - Assembler then processes that to generate object code *(executable binary code)*
  - ***Note:*** *The process consists of a number of stages – beyond the scope of this unit*

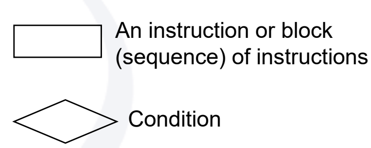


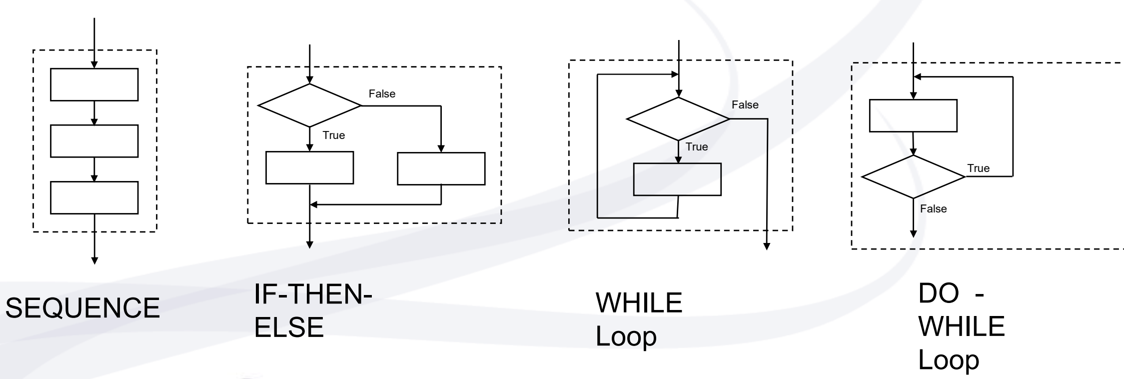
Program flow control

- Simplest program is a simple *straight-line* (sequential) program
- Execution is carried out on instructions stored sequentially in memory
  - limited to relatively simple programming tasks
- For more complex programs some form of *branching* (which includes looping) is usually required
- Branching allows the microprocessor to break out of the normal straight-line sequence
  - Jump or branch to a different section of code in program memory to continue executing instructions

Program flow control

- **High-level language constructs**





Jump and Branch Instructions

- ***Unconditional jump***
  - Processor will *always* jump to the new address given (e.g. **JMP**)
  - new address loaded into PC
    - *Refer addressing modes – Module 3*
- ***Conditional branch*** instructions
  - more flexibility in program flow control
  - will alter the sequence of execution only when specific conditions are met
    - E.g. JZ, JNZ, JG, etc. *(refer conditional branch instructions – Module 4)*

Conditional Branch Instructions *(recap)*

- Most conditional branch instructions are designed to be used after a CMP or arithmetic operation between two operands
  - e.g. A and B
  - Action whether to branch (jump) or not based on particular flags

Mnemonic	Condition	Mnemonic	Condition
JA	A > B (unsigned)	JC	C flag = 1
JAE	A ≥ B (unsigned)	JNC	C flag = 0
JB	A < B (unsigned)	JNO	O flag = 0
JBE	A ≤ B (unsigned)	JNP	P flag = 0
JE	A = B (Z = 1)	JNS	S flag = 0
JG	A > B (signed)	JNZ	Z flag = 0
JL	A < B (signed)	JO	O flag = 1
JNA	A ≯ B (unsigned)	JPE	P flag = 1
JNE	A ≠ B	JS	S flag = 1
JNL	A ≮ B	JZ	Z flag = 1

Absolute program addressing *(recap Module 3)*

- Also known as *direct* mode
- The instruction specifies a value that is written directly to the PC register

  - *Intel 8086* example
    - JMP 02A3H ; Program to **Jump** to location 02A3h

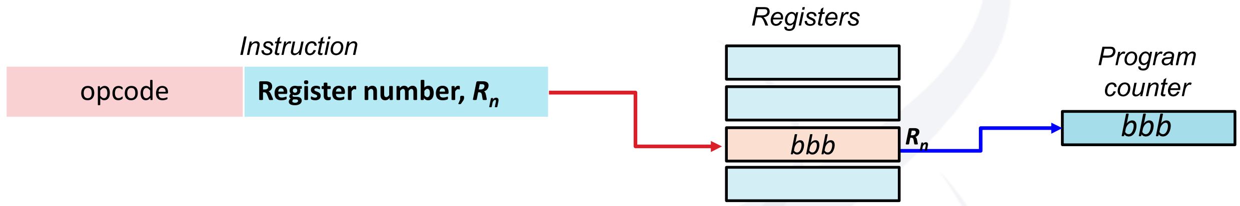
Relative program addressing *(recap Module 3)*

- The instruction specifies a value (positive or negative) that is added to the PC register
  - Causes the program to move forward or backward a certain number of bytes

  - *Intel 8086* example
    - JNZ 08 ; **Jump if** previous result is **Not Zero** forward 8 bytes

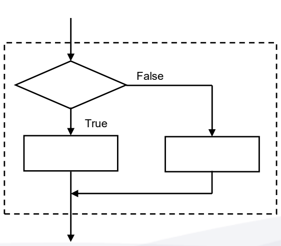
Indirect program addressing *(recap Module 3)*

- The program counter is loaded with the value from the register specified in the instruction

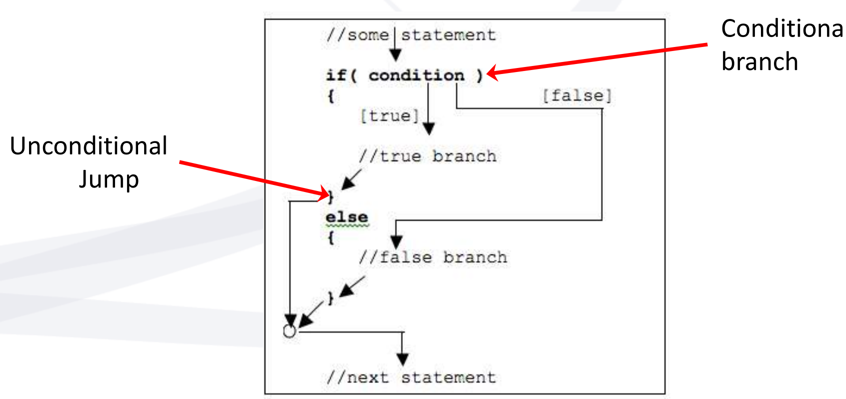


IF .. THEN .. ELSE

- IF condition = TRUE : THEN branch executed
- IF condition = FALSE : ELSE branch executed
- One conditional branch and one unconditional jump instruction required

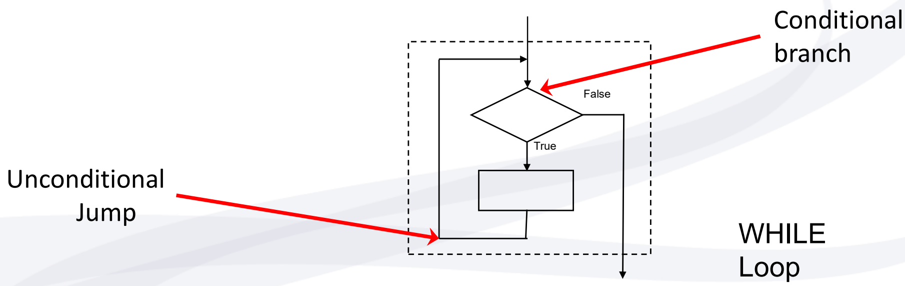


IF-THEN-ELSE



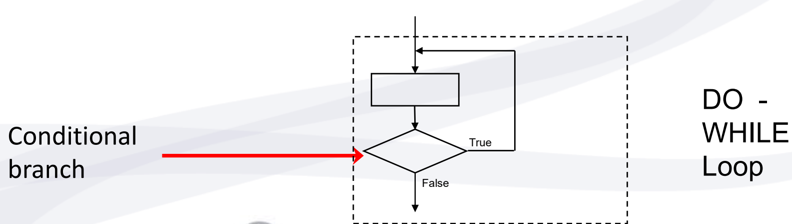
WHILE Loop

- Condition tested at the start of the loop
- IF condition = True THEN execute the loop code ELSE exit the loop
  - Similar to IF statement except that unconditional jump is back to the condition check
- Code in the loop is never run if condition is false the first time checked



DO – WHILE Loop

- Condition tested at the end of the loop
- IF condition = True THEN loop back to start (*DO*)
  ELSE continue next instruction (*exit the loop*)
  - Only requires 1 conditional branch instruction
  - Condition at the end of the loop
- The loop code will run at least once
  - even if condition is false the first time checked



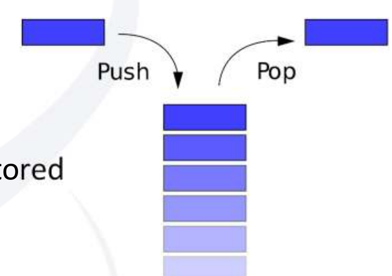
Assembler features for flow control

- **Symbolic Addresses (Labels)**
  - One of the most useful features of assembly language programming:
    - symbolic addresses may be used
    - instead of physical RAM addresses

Address	Label	Mnemonic	Comments
0100	**Start:**	CMP AX, BX	; Compare registers AX and BX
0101		JNE *Here*	; Jump to label 'Here' if not equal ; (difference not 0)
0102		JMP *FarAway*	; Otherwise jump to label 'Faraway'
0103	**Here:**	……	; Code to be done if not equal
…	…	…	…
0200	**Faraway:**		; Some code fragment elsewhere for 'ELSE'

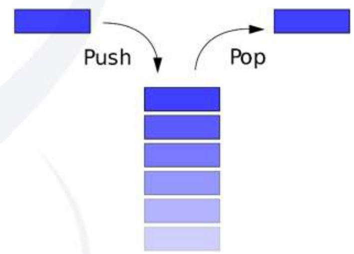
Stack

- A data structure (memory locations)
  - Primarily for temporary storage of information
- Storing ('pushing') a word
  - puts it at "top" of stack
  - location address is one word less than the previous word stored
- Words must be read ('popped') from the top of stack
  - in the opposite order from that in which they were placed
- The stack is referred to as being *last in, first out* or LIFO
- The address of the top of the stack is stored in the Stack Pointer register (SP)



Stack

- An important concept in programming
- Used by programmers :
  - Temporary storage of register contents etc.
- Used by the processor :
  - Storage of return addresses and other critical information
  - During subroutine calls *(covered next)*
  - Also for interrupt service routines *(covered in Module 8)*

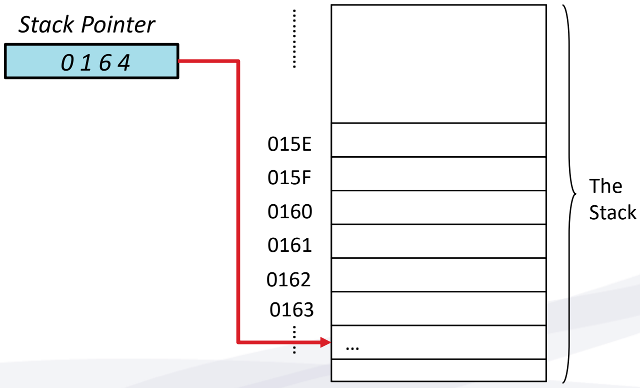


Stack Instructions

- **8086 Stack Instructions**
  - The processor controls the stack pointer (SP) during these operations
- **PUSH** *Reg / Mem*
  - Decrements SP by 2 *(stores 16-bit values, 2 bytes)*
    - Moves pointer up to new 'top of stack'
  - Copies contents of the specified register or memory location onto the stack
- **POP** *Reg / Mem*
  - Copies data from the top of the stack (pointed to by SP)
  - Then loads the data into the specified register or memory location
  - Increments SP by 2
    - Moves pointer down to new 'top of stack'

Pushing onto the stack

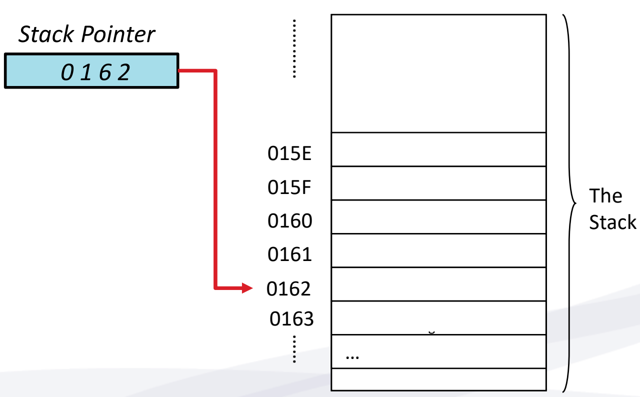
- Example



- 3 words stored onto the stack:
  - Word A
  - then B
  - finally Word C
- Initially SP = address 0164h

Pushing onto the stack

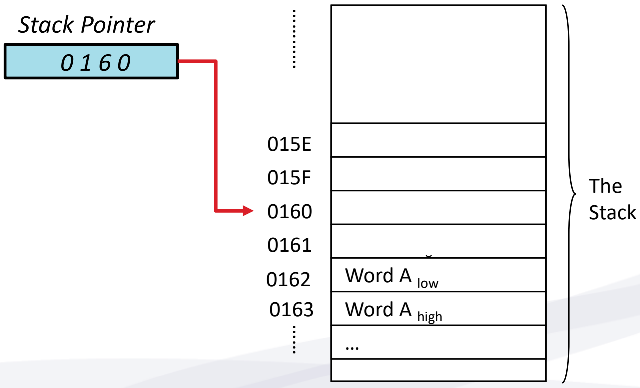
- Example



- 3 words stored onto the stack:
  - Word A
  - then B
  - finally Word C
- Initially SP = address 0164h
- When Word A pushed:
  - SP automatically decremented to point to 0162h
  - Then Word A stored (*little endian*)

Pushing onto the stack

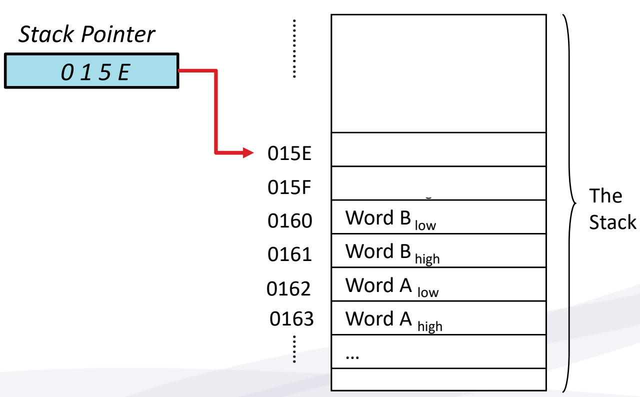
- **Example**



- 3 words stored onto the stack:
  - Word A
  - then B
  - finally Word C
- Initially SP = address 0164h
- When Word A pushed:
  - SP automatically decremented to point to 0162h
  - Then Word A stored (*little endian*)
- Then Word B

Pushing onto the stack

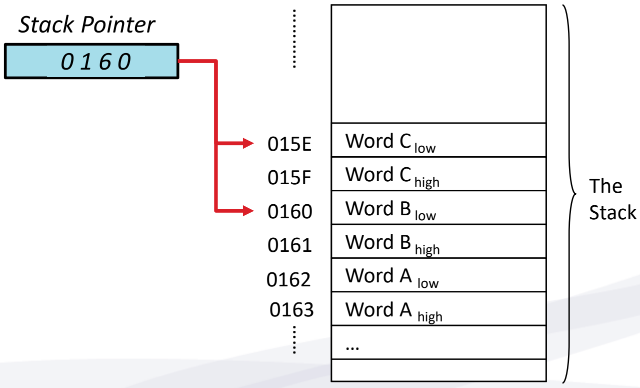
- **Example**



- 3 words stored onto the stack:
  - Word A
  - then B
  - finally Word C
- Initially SP = address 0164h
- When Word A pushed:
  - SP automatically decremented to point to 0162h
  - Then Word A stored (*little endian*)
- Then Word B
- After Word C pushed:
  - SP will contain the address 015Eh
  - the new "top" of the stack
- Reverse process happens when 'popping off' data from stack.

Popping off the stack

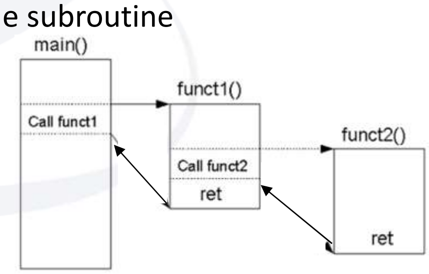
- **Example**



- Reverse process happens when ‘popping off’ data from stack.
  - E.g POP *AX*
    - Top of stack data copied to register AX
      - Then SP automatically incremented to point to 0160h
      - the new “top” of the stack

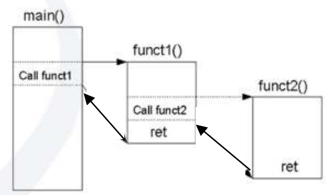
Subroutines

- **What is a Subroutine?**
  - A subroutine:
    - is a sequence of instructions that perform a specific task
    - it is written once and stored in a specific area of memory
  - When main program needs to perform that task:
    - the processor simply jumps to the address for the subroutine (*call*)
    - executes the code there
    - returns to original program when done



Calling a subroutine

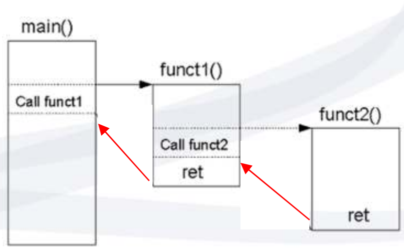
- Most processors have a **CALL** Instruction
  - Used to direct the processor to jump to a subroutine
  - Name may vary, but function is essentially the same
    - Operand provided gives the subroutine address to go to
    - The subroutine address is loaded into the program counter (PC)
    - Execution of subroutine continues as normal till the end of the routine
    - Program execution returns to calling routine (*next instruction*)



- How does processor now how to go back to *calling* routine?
  - The **PC** register contents are pushed onto the stack before new subroutine address is loaded

Returning from a subroutine

- Most processors have a Return (**RET**) instruction
  - Used to direct the processor to return to the calling process
    - The RET instruction pops the return address back from the top of the stack into the PC register
    - Continues with next instruction after the initial CALL



Subroutines

- **Example using assembly language**

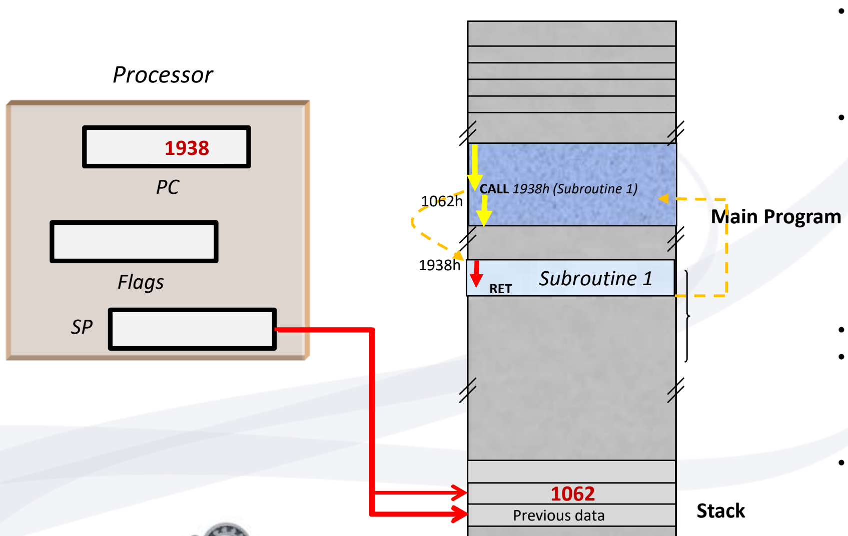
```
...
CALL NewProcedure          ; call procedure
 xxx                       ; next instruction
....
 ....

NewProcedure PROC NEAR     ; Subroutine to do something
      xxxx
      ...
      RET                  ; Return to location called from (next inst)
NewProcedure ENDP
```

Push PC (address of instruction *xxx*) on stack, load PC with *NewProcedure* address

Pop PC from stack (address of instruction *xxx* pushed previously)

Subroutine Calls



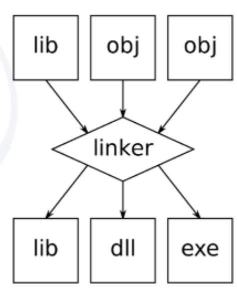
- User program runs
- Fetches 'CALL *Subroutine 1'*
  - PC is incremented to 0162h (ready for next instruction
- Executes the CALL instruction
  - PC contents pushed to stack
  - Address of Subroutine 1 loaded from program memory into PC
- Subroutine runs
- *Return* instruction reached and executed
  - Saved PC value popped from stack
- Main program continues from after subroutine call

Why use subroutines?

- Allows for efficient and logical coding
  - Commonly used functions can be embedded in subroutines and called repeatedly as needed
  - Do need to repeat the code for a function over and over in the main program
  - Once tested, can be sure the routine works – less errors
- Allows for previously developed code to be used
  - E.g. library functions
- Allows code written in a different platform / language to be linked
  - Provided interface / parameter passing is correct
  - E.g. calling an assembly routine from a C program

Linker

- A linker is a program that combines various object files into a single executable file
  - A program may call many subroutines that sit in various libraries
  - The linker pulls in the required (called) library functions and creates a single executable (machine code) file that has all the required code.
- Linkers may also relocate code
  - Change the addresses of absolute jumps, load and stores
  - Based on where a program is loaded in memory



Assemblers, Compilers and Linkers

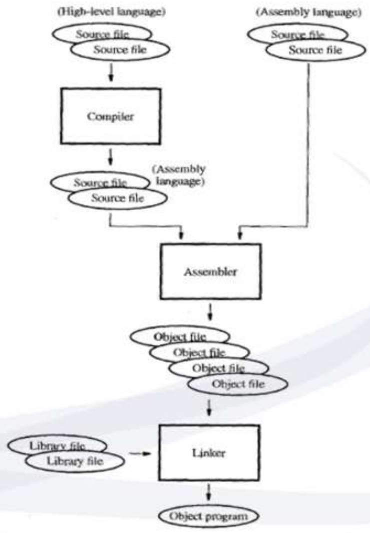


Figure 4.1 Overall flow for generating an object program.

(Hamacher, 2016)

Module Objectives

On completion of this module, students should be able to:

- Explain the differences between high-level languages (HLLs) and assembly language, and the advantages and disadvantages of writing programs in each of these.
- Describe the process of translating programs into executable binary code, including the translator tools used.
- Explain how HLL program control structures are implemented using conditional branch instructions.
- Explain what a stack is and how it functions.
- Explain how subroutine calls and returns work, including the role of the stack in these processes.