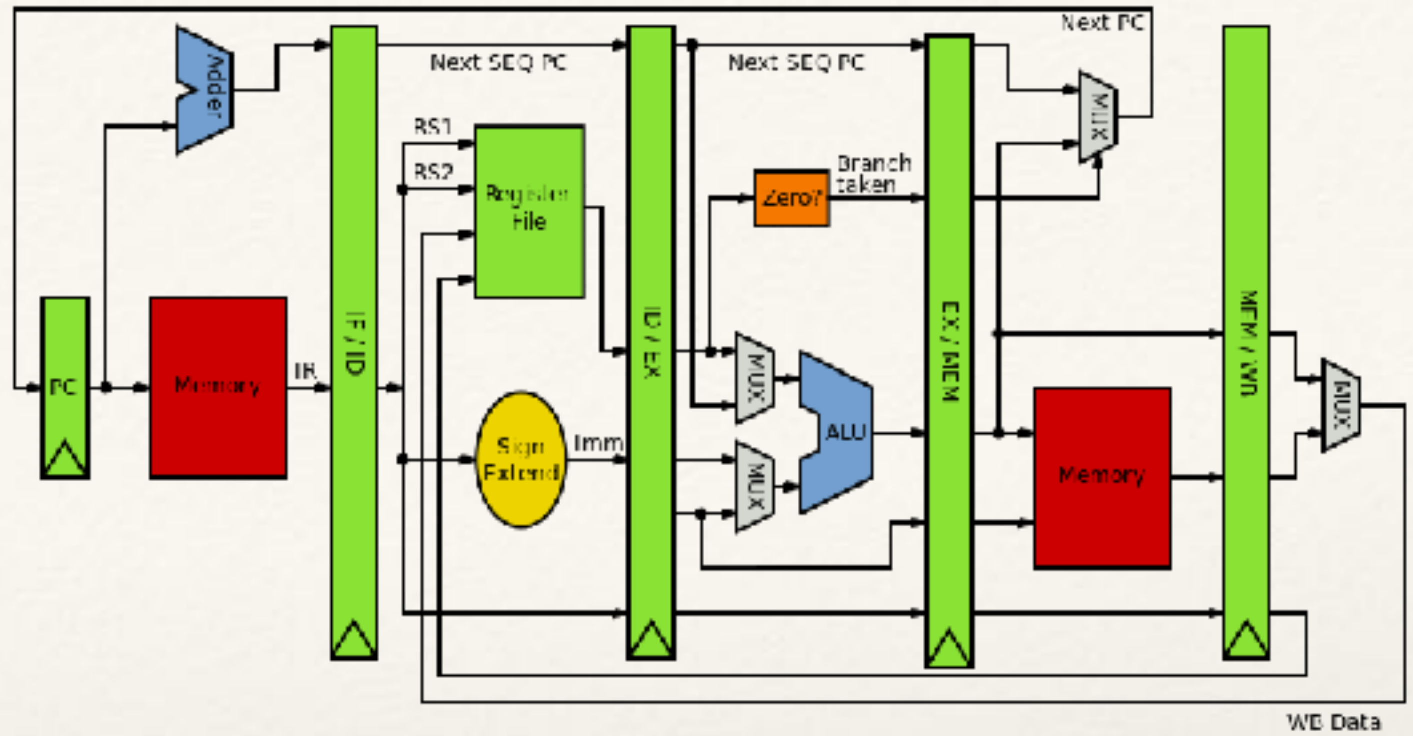


COMP2120 Computer Organization

Instruction Sets

Chapter 12, 13, and Appendix B

Dr. Ronald H.Y. Chung

THE UNIVERSITY OF HONG KONG
DEPARTMENT OF
COMPUTER SCIENCE

Machine Instruction Characteristics

- The operation of the processor is determined by the instructions it executes, referred to as ***machine instructions*** or ***computer instructions***
- The collection of different instructions that the processor can execute is referred to as the processor's ***instruction set***
- Each instruction must contain the information required by the processor for execution

Instructions Types

Data processing

- Arithmetic instructions provide computational capabilities for processing numeric data
- Logic (Boolean) instructions operate on the bits of a word as bits rather than as numbers, thus they provide capabilities for processing any other type of data the user may wish to employ

Data storage

- Movement of data into or out of register and or memory locations

Control

- Test instructions are used to test the value of a data word or the status of a computation
- Branch instructions are used to branch to a different set of instructions depending on the decision made

Data movement

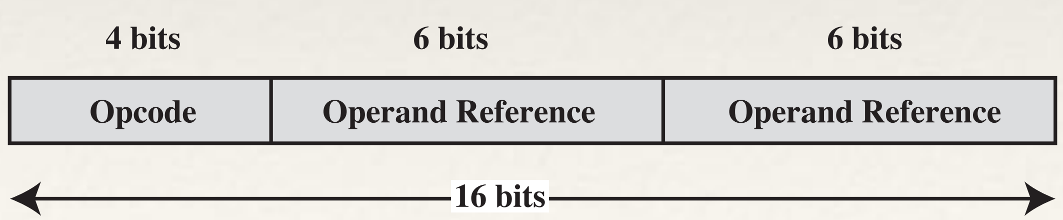
- I/O instructions are needed to transfer programs and data into memory and the results of computations back out to the user

Elements of a Machine Instruction

- Operation code
  - Specifies the operation to be performed (e.g., ADD, I/O). The operation is specified by a binary code, known as the operation code, or opcode.
- Source operand reference
  - The operation may involve one or more source operands, that is, operands that are inputs for the operation.
- Result operand reference
  - The operation may produce a result.
- Next instruction reference
  - This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

Instruction Representation

- Within the computer each instruction is represented by a sequence of bits
- The instruction is divided into fields, corresponding to the constituent elements of the instruction
- Opcodes are represented by abbreviations called ***mnemonics***
- Examples include:
    - ADD Add
    - SUB Subtract
    - MUL Multiply
    - DIV Divide
    - LOAD Load data from memory
    - STOR Store data to memory
    - CALL/JMP Call Function/Jump Subroutine
- Operands are also represented symbolically
- Each symbolic operand has a fixed binary representation
    - The programmer specifies the location of each symbolic operand



Example Instruction Format

Common Instruction Set Operations

Type	Operation Name	Description
Data Transfer	Move (transfer)	Transfer word or block from source to destination
	Store	Transfer word from processor to memory
	Load (fetch)	Transfer word from memory to processor
	Exchange	Swap contents of source and destination
	Clear (reset)	Transfer word of 0s to destination
	Set	Transfer word of 1s to destination
	Push	Transfer word from source to top of stack
	Pop	Transfer word from top of stack to destination
Arithmetic	Add	Compute sum of two operands
	Subtract	Compute difference of two operands
	Multiply	Compute product of two operands
	Divide	Compute quotient of two operands
	Absolute	Replace operand by its absolute value
	Negate	Change sign of operand
	Increment	Add 1 to operand
	Decrement	Subtract 1 from operand
Logical	AND	Perform logical AND
	OR	Perform logical OR
	NOT (complement)	Perform logical NOT
	Exclusive-OR	Perform logical XOR
	Test	Test specified condition; set flag(s) based on outcome
	Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
	Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
	Shift	Left (right) shift operand, introducing constants at end
	Rotate	Left (right) shift operand, with wraparound end

Type	Operation Name	Description
Transfer of Control	Jump (branch)	Unconditional transfer; load PC with specified address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued
Input/Output	Input (read)	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
	Output (write)	Transfer data from specified source to I/O port or device
	Start I/O	Transfer instructions to I/O processor to initiate I/O operation
	Test I/O	Transfer status information from I/O system to specified destination
Conversion	Translate	Translate values in a section of memory based on a table of correspondences
	Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

Processor Actions for Various Types of Operations

Data Transfer	Transfer data from one location to another
	If memory is involved: Determine memory address Perform virtual-to-actual-memory address transformation Check cache Initiate memory read/write
Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes and flags
Logical	Same as arithmetic
Conversion	Similar to arithmetic and logical. May involve special logic to perform conversion
Transfer of Control	Update program counter. For subroutine call/return, manage parameter passing and linkage
I/O	Issue command to I/O module
	If memory-mapped I/O, determine memory-mapped address

Arithmetic vs Logical Operations

- Arithmetic operation
  - Treat the operands as numbers, and has to consider the sign of the operands, in particular, on shift operations. (in 2's complement)
    - Shift operations are equivalent to arithmetic operations
    - Shift Left — Multiply by 2
    - Shift Right — Divide by 2 (and remainder is shifted out)
    - Shift operations has to preserve the sign of the operands if it is arithmetic
- Logic Operation - treat the operands as bit patterns
- e.g. Arithmetic Shift Right has to preserve the sign, hence has to copy the original sign bit. (Not for logical shift operation)

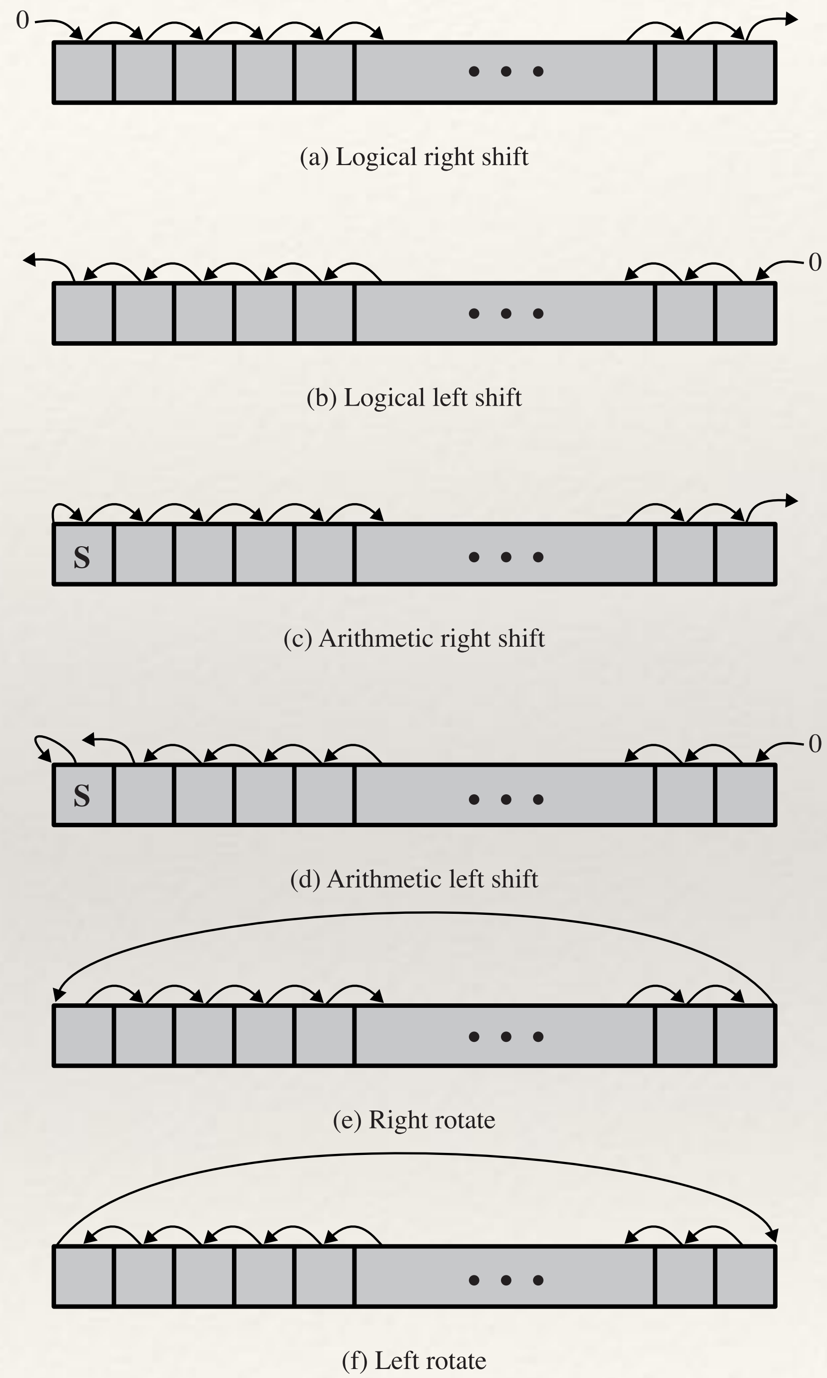


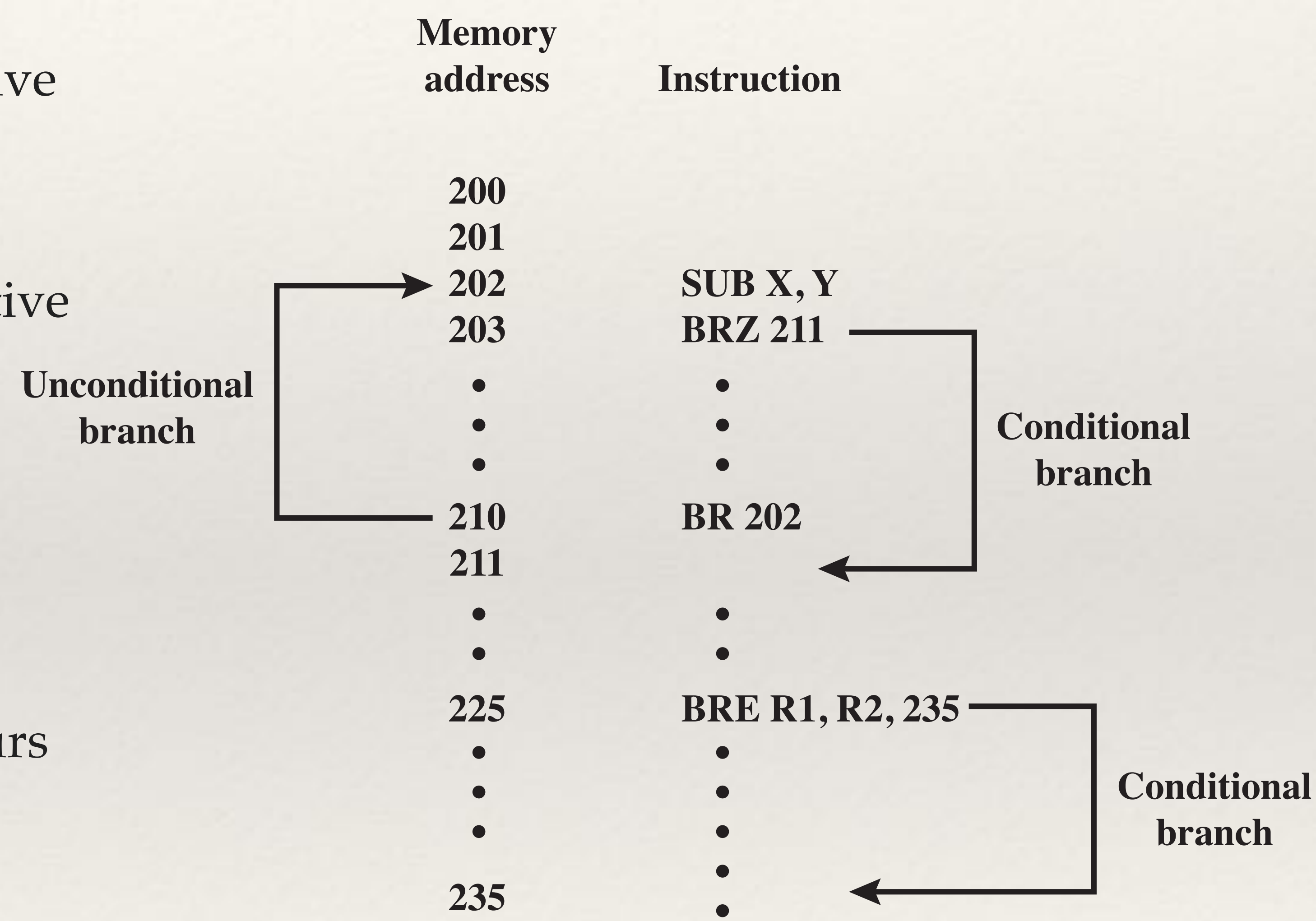
Figure 12.6 Shift and Rotate Operations

Transfer of Control

- Reasons why transfer-of-control operations are required:
  - It is essential to be able to execute each instruction more than once
  - Virtually all programs involve some decision making
  - It helps if there are mechanisms for breaking the task up into smaller pieces that can be worked on one at a time
- Most common transfer-of-control operations found in instruction sets:
  - Branch
  - Skip
  - Procedure call

Example Branch Instructions

- BRP X
  - Branch to location X if result is positive
- BRN X
  - Branch to location X if result is negative
- BRZ X
  - Branch to location X if result is zero
- BRO X
  - Branch to location X if overflow occurs
- BRE A, B, X
  - Branch to location X if A equals B



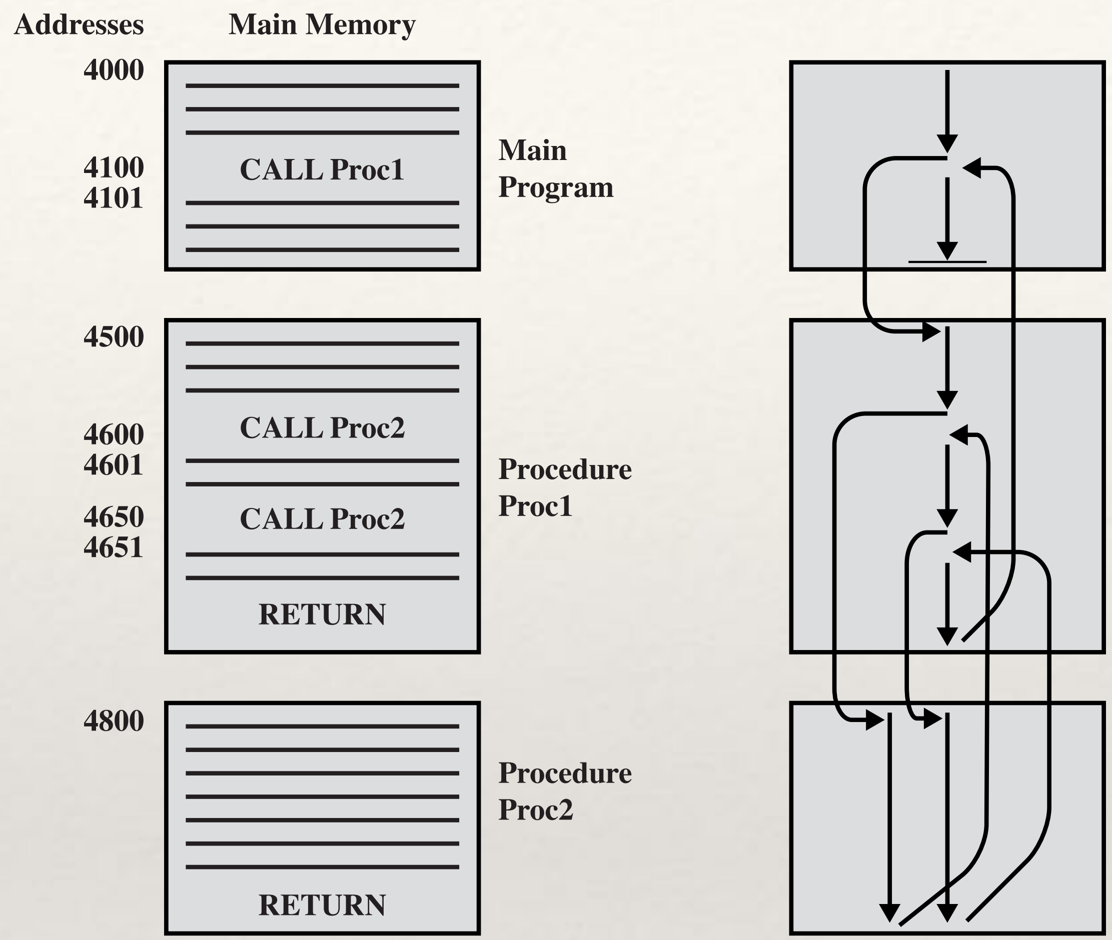
SKIP Instruction

- The skip instruction includes an implied address
  - Implied address equals the address of the next instruction plus one instruction length
- Typical example
  - ISZ - Increment-and-skip-zero
    - 301
    - .
    - .
    - 309 ISZ R1
    - 310 BR 301
    - 311
  - R1 is set with negative of the number of iterations to be performed, at the end of the loop, R1 is incremented
    - Branch instruction is skipped when R1 is zero

Procedure Call Instructions

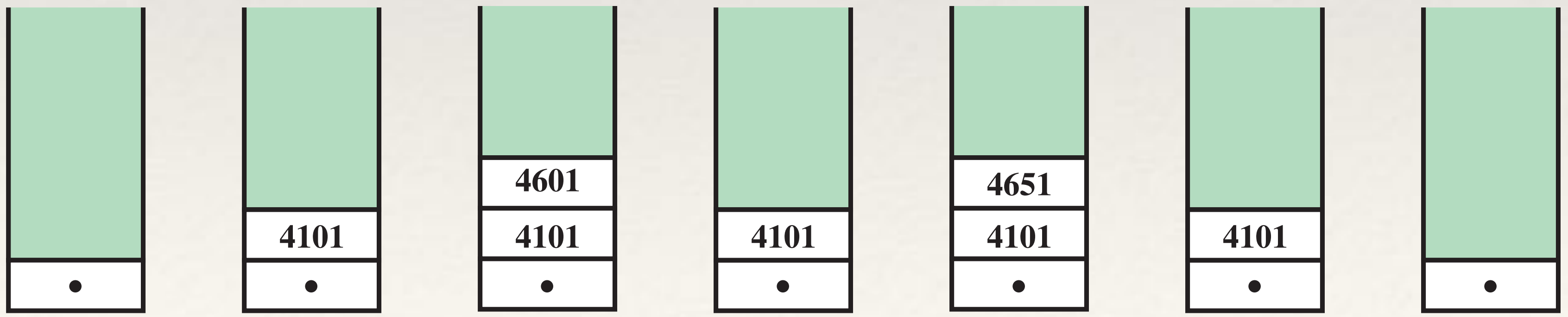
- Self-contained computer program that is incorporated into a larger program
  - At any point in the program the procedure may be invoked, or called
  - Processor is instructed to go and execute the entire procedure and then return to the point from which the call took place
- Two principal reasons for use of procedures:
  - Economy
    - A procedure allows the same piece of code to be used many times
  - Modularity
- Involves two basic instructions:
  - A call instruction that branches from the present location to the procedure
  - Return instruction that returns from the procedure to the place from which it was called
  - The return addresses are retrieved in a Last-in-First-Out fashion
    - Return address can be stored in the system stack

Nested Procedures



(a) Calls and returns

(b) Execution sequence



(a) Initial stack contents

(b) After CALL Proc1

(c) Initial CALL Proc2

(d) After RETURN

(e) After CALL Proc2

(f) After RETURN

(g) After RETURN

Number of Operands

- Number of operand for most instruction is typically three — two source (src) and one destination (dst) operand.
- Depending on the design of instruction set, operands can be *implied*
  - 3 operand instruction — used in most instruction set nowadays (the one used in asg 2).
    - e.g. ADD A, B, C — C←A+B
  - 2 operand instruction — one of the src operand also acts as the dst operand.
    - e.g. ADD A, B — B←A+B
  - 1 operand instruction — a default operand which act as both src and dst (usually called the accumulator AC, hence called accumulator-based machine)
    - e.g. ADD A — AC←AC+A
  - 0 operand instruction — all instruction are implied. They are usually on the stack, i.e. Stack machine.
    - e.g. ADD — Pop top 2 elements from stack and ADD, then push it back.

Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	A ← B OP C
2	OP A, B	A ← A OP B
1	OP A	AC ← AC OP A
0	OP	T ← (T – 1) OP T

AC = accumulator
T = top of stack
(T – 1) = second element of stack
A, B, C = memory or register locations

Number of Operands in Instructions

Instruction		Comment
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	$Y \leftarrow A$
SUB	Y, B	$Y \leftarrow Y - B$
MOVE	T, D	$T \leftarrow D$
MPY	T, E	$T \leftarrow T \times E$
ADD	T, C	$T \leftarrow T + C$
DIV	Y, T	$Y \leftarrow Y \div T$

(b) Two-address instructions

Instruction		Comment
LOAD	D	$AC \leftarrow D$
MPY	E	$AC \leftarrow AC \times E$
ADD	C	$AC \leftarrow AC + C$
STOR	Y	$Y \leftarrow AC$
LOAD	A	$AC \leftarrow A$
SUB	B	$AC \leftarrow AC - B$
DIV	Y	$AC \leftarrow AC \div Y$
STOR	Y	$Y \leftarrow AC$

(c) One-address instructions

Programs to Execute $\mathbf{Y = \dfrac{A - B}{C + (D \times E)}}$

Registers

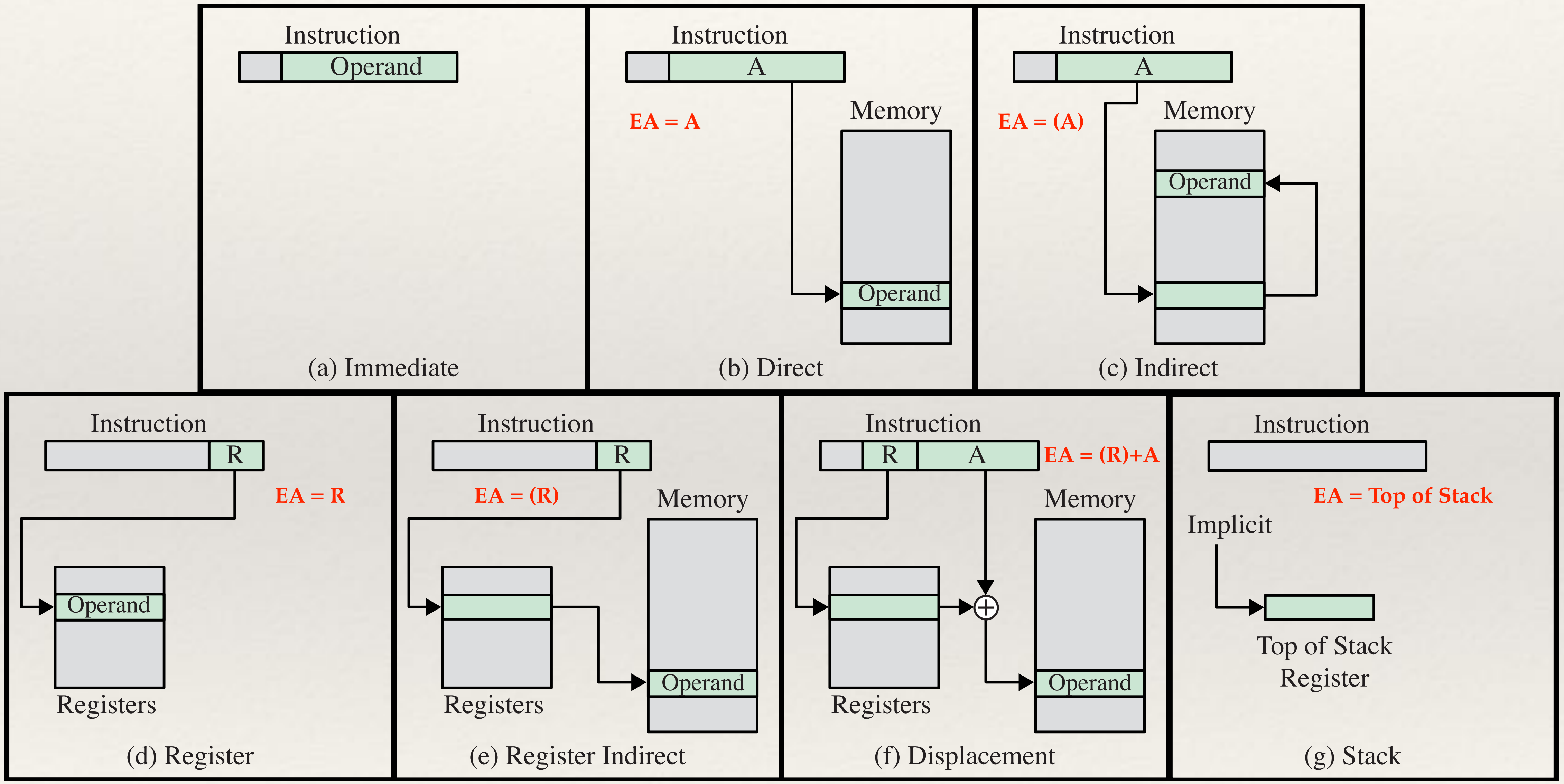
- General Purpose Registers vs Dedicated Purpose Registers
  - general purpose registers — registers can be used freely.
  - special purpose registers — registers have dedicated purpose, for example, to point to strings, as operands for floating point instructions etc.
  - There are always some dedicated purpose registers, such as Program Counter (PC) and Stack Pointer (SP).
- There is a special register called PSW (processor status word) or flag register which define some condition codes

Addressing Mode

- Describe the way to calculate the (effective) address of an operand.
- Built into the instruction set of a machine, no need to use extra instructions.
- Using addressing mode can reduce the size of program code, because no code is required for address calculation if the addressing mode is supported.
- However, the hardware will be more complicated.
- Modern processors provide a large number of registers. Hence the need to reference memory is reduced.
- Also, according to statistics, only a few addressing mode is widely used.
- Hence the number of addressing modes available in modern processor is very small.

Addressing Mode

Note: EA = Effective Address



Address Mode Comparison

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Register Indirect Addressing Example

- The address of the operand is in the specified register (pointer like).
- Most assembly language specify this mode by (R1), or @R1.
- Example: (an assembly lang prog to find the sum of an array)
  - Assume a 3-operand instruction set:

```
        XOR     R1, R1, R1      ; init R1 to 0
        MOV     #A, R2          ; R2 = add of array A
        MOV     #1024, R3       ; R3 = size of array
Loop:   ADD     R1, (R2), R1    ; R1 = R1 + (R2)
        ADD     R2, #4, R2      ; incr R2 to next elt
        SUB     R3, #1, R3      ; decrement R3
        BNE     LOOP            ; if result != 0 branch
```

Assembly Language Programming

- We shall not focus on any CPU. Instead, we invent a simple instruction set for a hypothetical machine
- Assembly Language Elements

Label	Mnemonic	Operand(s)	;comment
Optional	Opcode Name	Zero or more	Optional
	or		
	Directive Name		
	or		
	Macro Name		

Assembler Directives

Directives	Description
.data	Tells the assembler to add all subsequent data to the data section.
.text	Tells the assembler to add subsequent code to the text section (i.e. program section)
.global name	Makes name external to other files, for multiple files in the program
.space expression	Reserves spaces, amount specified by the value of the expression in bytes. The assembler fills the space with zeros
.word exp1 [, exp2] … [, expN]	Put the values in successive memory locations

ALP - Control Structures

- Example:

  if (a[0] > a[1]) x = a[0] ;

  else x = a[1] ;

```
        .data                   ; data segment
a:      .word   1               ; create storage containing 1
        .word   3               ; create storage containing 3
x:      .word   4               ; create storage containing 4
        .text
main:   ld      #a, r8          ; r8 = address of a
        ld      0(r8), r9       ; r9 = a[0]
        ld      4(r8), r10      ; r10 = a[1]
        bgt     r9, r10, f1     ; if r9>r10, goto f1
        st      r10, x          ; x = R10
        br      f2              ; goto f2
f1:     st      r9, x           ; x = r9
f2:     ret                     ; return to OS
```

ALP - Repetition Construct

- Example:

  a = 0 ;

  for (i = 0; i < 10; i++) a += i ;

```
            .data
a:          .word       0
            .text
main:
            sub         r8, r8, r8          ; r8 = 0
            ld          #0xa, r9            ; r9 = 10, no. of iterations
            sub         r10, r10, r10       ; r10 = 0, loop counter
            ld          #1, r11             ; load immediate, r11=1
f1:         add         r8, r10, r8         ; r8 += r10
            add         r10, r11, r10       ; r10++, increment counter
            bne         r9, r10, f1         ; if(r9 != r10) goto f1
            st          r8, a               ; a = r8
            ret                             ; return
```

ALP - While Construct

- Example:

```
temp = 0 ;
a = 1 ;
while (temp < 100) {
    temp += a ;
    a++ ;
}
```

```
         ; fill in the .data, .text stuffs as in previous examples
         sub      r8, r8, r8       ; r8 is temp, r8 = 0
         ld       #1, r9           ; r9 is a, r9 = 1
         mv       r9, r10          ; r10 = r9
         ld       #0x64, r11       ; r11 = 100
f1:      add      r8, r9, r8       ; temp += a
         add      r9, r10, r9      ; a++
         blt      r8, r11, f1      ; if (r8 < r11) goto f1
```

Function Calls

- Use ***call*** and ***ret*** for function calls. Return address stored in the stack
- You need to specifies the input parameters and the output parameters for your function. They may be put in registers or memory
- For example, two input parameters stored in r8 and r9, and return parameter in r10
- If you change any values of the registers in the function, you need either to spell it out in the program specification, or push the original value, and pop it out at the end of the function call

Function Call Example

- Example:

  convert all characters into upper case letters

```
            .data
a:          .asciiz    “This is a test” ; zero-terminated string
            .text

main:       sub        r9, r9, r9       ; r9 = 0
loop:       lb         a(r9), r10       ; load byte
            beq        r10, #0, exit    ; r10 == 0 ? end of string
            call       capitalize       ; call capitalise function
            sb         r10, a(r9)       ; store result back
            add        r9, 1, r9        ; increment r9, next char
            br         loop             ; goto loop
exit:       ret                         ; return
```

Function Call Example (Cont'd)

- Example:

  convert all characters into upper case letters

```
Capitalize:
;input is r10, output is r10, if r10 is lower case letter
;  change to upper case

          push      r8
          push      r9
          ld        #0x61, r8         ; r8 = 'a'
          ld        #0x7a, r9         ; r9 = 'z'
          blt       r10, r8, ret1
          bgt       r10, r9, ret1
          sub       r10, #0x20, r10   ; 0x20 = 'a'-'A'
ret1:     pop       r9
          pop       r8
          ret                         ; return
```

Summary

- Machine instruction characteristics
  - Elements of a machine instruction
  - Instruction representation
  - Instruction types
  - Number of addresses
  - Instruction set design
- Assembly Programming
  - Assembly Directives
  - Control Structures
  - Function Calls
- Addressing Modes
  - Addressing modes
  - Immediate addressing
  - Direct addressing
  - Indirect addressing
  - Register addressing
  - Register indirect addressing
  - Displacement addressing
  - Stack addressing