İstanbul
Bilgi Üniversitesi

CMPE321 - Computer Architecture

Lecture 9
{3,2,1,0}-Address Instructions

Hakan Ayral, PhD.

Three-Address Instructions

- A program that evaluates $X = (A + B)(C + D)$ using three-address instructions is as follows (a register transfer statement is shown for each instruction):

ADD T1, A, B	$M[T1] \leftarrow M[A] + M[B]$
ADD T2, C, D	$M[T2] \leftarrow M[C] + M[D]$
MUL X, T1, T2	$M[X] \leftarrow M[T1] \times M[T2]$

- The symbol ***M*[*A*]** denotes the operand stored in memory at the address symbolized by **A**.
- The symbol × designates multiplication.
- **T1** and **T2** are temporary storage locations in memory.

Three-Address Instructions

- This same program can use registers as the temporary storage locations:

ADD R1, A, B	$R1 \leftarrow M[A] + M[B]$
ADD R2, C, D	$R2 \leftarrow M[C] + M[D]$
MUL X, R1, R2	$M[X] \leftarrow R1 \times R2$

- Use of registers reduces the data memory accesses required from nine to five.
- An advantage of the three-address format is that it results in short programs for evaluating expressions.
- A disadvantage is that the binary-coded instructions require more bits to specify three addresses, particularly if they are memory addresses.

Two-Address Instructions

- For two-address instructions, each address field can again specify either a possible register or a memory address.
- The first operand address listed in the symbolic instruction also serves as the implied address to which the result of the operation is transferred.
- The program is as follows:

MOVE T1, A	$M[T1] \leftarrow M[A]$
ADD T1, B	$M[T1] \leftarrow M[T1] + M[B]$
MOVE X, C	$M[X] \leftarrow M[C]$
ADD X, D	$M[X] \leftarrow M[X] + M[D]$
MUL X, T1	$M[X] \leftarrow M[X] \times M[T1]$

- If a temporary storage register **R1** is available, it can replace **T1**.
- Note that this program takes five instructions instead of the three used by the three-address instruction program.

One-Address Instructions

- To perform instructions such as **ADD**, a computer with one-address instructions uses an implied address—such as a register called an ***accumulator, ACC***—for obtaining one of the operands and as the location of the result.
- The program to evaluate the arithmetic statement is as follows:

LD	A	$ACC \leftarrow M[A]$
ADD	B	$ACC \leftarrow ACC + M[B]$
ST	X	$M[X] \leftarrow ACC$
LD	C	$ACC \leftarrow M[C]$
ADD	D	$ACC \leftarrow ACC + M[D]$
MUL	X	$ACC \leftarrow ACC \times M[X]$
ST	X	$M[X] \leftarrow ACC$

- All operations are done between the ***ACC*** register and a memory operand.
- In this case, the number of instructions in the program has increased to seven and the number of memory data accesses is also seven.

Zero-Address Instructions

- To perform an **ADD** instruction with zero addresses, all three addresses in the instruction must be implied.
- A conventional way of achieving this goal is to use a *stack*, which is a mechanism or structure that stores information such that the item stored last is the first retrieved.
- Because of its "last-in, first-out" nature, a stack is also called a ***last-in, first-out*** (***LIFO***) queue.
- Data-manipulation operations such as **ADD** are performed on the stack.
- The word at the top of the stack is referred to as **TOS**.
- The word below it is **TOS1**.
- When one or more words are used as operands for an operation, they are removed from the stack.
- The word below them then becomes the new **TOS**.
- When a resulting word is produced, it is placed on the stack and becomes the new **TOS**.
- Thus, **TOS** and a few locations below it are the implied addresses for operands, and **TOS** is the implied address for the result.

Zero-Address Instructions

- For example, the instruction that specifies an addition is simply

ADD

- The resulting register transfer action is $TOS \leftarrow TOS + TOS_{-1}$.
- Thus, there are no registers or register addresses used for data-manipulation instructions in a stack architecture.
- Memory addressing, however, is used in such architectures for data transfers.
- For instance, the instruction

PUSH X

- results in , a transfer of the word in address X in memory to the top of the stack.
- A corresponding operation,

POP X

- results in , a transfer of the entry at the top of the stack to address **X** in memory.

Zero-Address Instructions

- The program for evaluating the sample arithmetic statement for the zero-address situation is as follows:

PUSH	A	$TOS \leftarrow M[A]$
PUSH	B	$TOS \leftarrow M[B]$
ADD		$TOS \leftarrow TOS + TOS_{-1}$
PUSH	C	$TOS \leftarrow M[C]$
PUSH	D	$TOS \leftarrow M[D]$
ADD		$TOS \leftarrow TOS + TOS_{-1}$
MUL		$TOS \leftarrow TOS \times TOS_{-1}$
POP	X	$M[X] \leftarrow TOS$

- This program requires eight instructions—one more than the number required by the previous one-address program.
- However, it uses addressed memory locations or registers only for **PUSH** and **POP** and not to execute data-manipulation instructions involving **ADD** and **MUL**.

Zero-Address Instructions

- Note that memory data accesses may be necessary, however, depending upon the stack implementation.
- Often, stacks utilize a fixed number of registers near the top of the stack.
- If a given program can be executed only within these stack locations, memory data accesses are necessary for fetching the initial operands and storing the final result only.
- But, if the program requires more temporary, intermediate storage, additional data accesses to memory are required.

Addressing Architectures

- The programs just presented change if the number of addresses to the memory in the instructions is restricted or if the memory addresses are restricted to specific instructions.
- These restrictions, combined with the number of operands addressed, define addressing architectures.
- We can illustrate such architectures with the evaluation of an arithmetic statement in a three-address architecture that has all of the accesses to memory.
- Such an addressing scheme is called a ***memory-to-memory architecture***.
- This architecture has only control registers, such as the program counter in the CPU.
- All operands come directly from memory, and all results are sent directly to memory.
- The formats of data transfer and manipulation instructions contain from one to three address fields, all of which are used for memory addresses.
- For the previous example, three instructions are required, but if an extra word must appear in the instruction for each memory address, then up to four memory reads are required to fetch each instruction.
- Including the fetching of operands and storing of results, the program to perform the arithmetic operation would require 21 accesses to memory.
- If memory accesses take more than one clock cycle, the execution time would be in excess of 21 clock periods.
- Thus, even though the instruction count is low, the execution time is potentially high.
- Also, providing the capability for all operations to access memory increases the complexity of the control structures and may lengthen the clock cycle.
- Thus, this memory-to-memory architecture is typically not used in new designs.

Addressing Architectures

- In contrast, the three-address ***register-to-register*** or ***load/store architecture***, which allows only one memory address and restricts its use to load and store types of instructions, is typical in modern processors.
- Such an architecture requires a sizeable register file, since all data manipulation instructions use register operands.
- With this architecture, the program to evaluate the sample arithmetic statement is as follows:

LD	R1, A	$R1 \leftarrow M[A]$
LD	R2, B	$R2 \leftarrow M[B]$
ADD	R3, R1, R2	$R3 \leftarrow R1 + R2$
LD	R1, C	$R1 \leftarrow M[C]$
LD	R2, D	$R2 \leftarrow M[D]$
ADD	R1, R1, R2	$R1 \leftarrow R1 + R2$
MUL	R1, R1, R3	$R1 \leftarrow R1 \times R3$
ST	X, R1	$M[X] \leftarrow R1$

Addressing Architectures

- Note that the instruction count increases to eight compared to three for the three-address, memory-to-memory case.
- Note also that the operations are the same as those for the stack case, except for the need for register addresses.
- By using registers, the number of accesses to memory for instructions, addresses, and operands is reduced from 21 to 18.
- If addresses can be obtained from registers instead of memory, as discussed in the next section, this number can be further reduced.

Addressing Architectures

- Variations on the previous two addressing architectures include three-address instructions and two-address instructions with one or two of the addresses to memory.
- The program lengths and number of memory accesses tend to be intermediate between the previous two architectures.
- An example of a two-address instruction with a single memory address allowed is

$$\text{ADD} \qquad \text{R1, A} \qquad R1 \leftarrow R1 + M[A]$$

- This ***register-memory*** type of architecture remains prevalent among the current instruction set architectures, primarily to provide compatibility with older software using a specific architecture.
- The program with one-address instructions illustrated previously gives the *single-accumulator architecture*.
- Since this architecture has no register file, its single address is for accessing memory.
- It requires 21 accesses to memory to evaluate the sample arithmetic statement.
- In more complex programs, significant additional memory accesses would be needed for temporary storage locations in memory.
- Because of its large number of memory accesses, this architecture is inefficient and consequently, is restricted to use in CPUs for simple, low-cost applications that do not require high performance.

Addressing Architectures

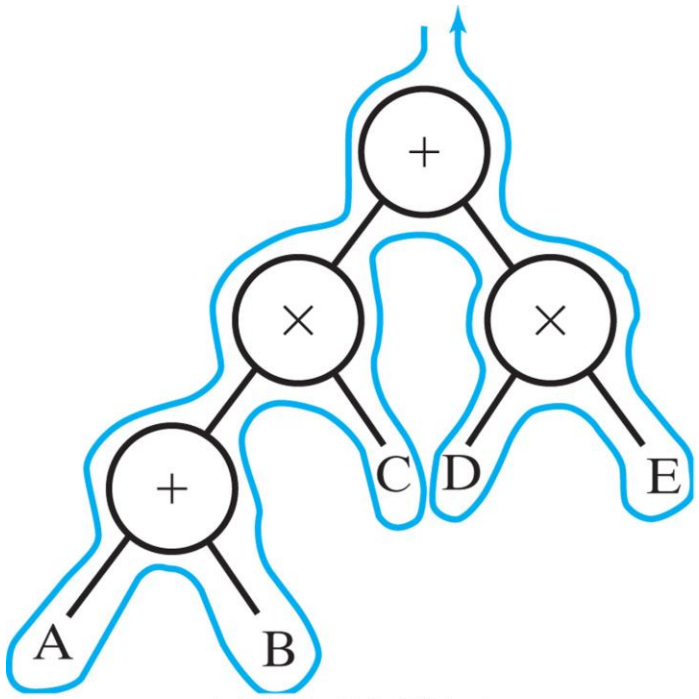
- The program with one-address instructions illustrated previously gives the ***single-accumulator architecture***.
- Since this architecture has no register file, its single address is for accessing memory.
- It requires 21 accesses to memory to evaluate the sample arithmetic statement.
- In more complex programs, significant additional memory accesses would be needed for temporary storage locations in memory.
- Because of its large number of memory accesses, this architecture is inefficient and consequently, is restricted to use in CPUs for simple, low-cost applications that do not require high performance.
- The zero-address instruction case using a stack supports the concept of a ***stack architecture***.
- Data-manipulation instructions such as ADD use no address, since they are performed on the top few elements of the stack.
- Single memory-address load and store operations, as shown in the program to evaluate the sample arithmetic statement, are used for data transfer.

Addressing Architectures

- Since most of the stack is located in memory, as discussed earlier, one or more hidden memory accesses may be required for each stack operation.
- As register-register and load/store architectures have made strong performance advances, the high frequency of memory accesses in stack architectures has made them unattractive.
- However, stack architectures have begun to borrow technological advances from these other architectures.
- These architectures store substantial numbers of stack locations in the processor chip and handle transfers between these locations and the memory transparently.
- Stack architectures are particularly useful for rapid interpretation of high-level language programs in which the intermediate code representation uses stack operations.

Addressing Architectures

- Stack architectures are compatible with a very efficient approach to expression processing which uses postfix notation rather than the traditional infix notation to which we are accustomed.
- The infix expression $(A + B) \times C + (D \times E)$ with the operators between the operands can be written as the postfix expression $A\ B + C \times D\ E \times +$
- Postfix notation is called reverse Polish notation (RPN), honoring the Polish mathematician Jan Lukasiewicz, who proposed prefix (the reverse of postfix) notation; prefix was also known as Polish notation.
- Conversion of (A + B) × C + (D × E) to RPN can be achieved graphically, as shown in Figure.
- When the path shown traversing the graph passes a variable, that variable is entered into the RPN expression.
- When the path passes an operation for the final time, the operation is entered into the RPN expression.



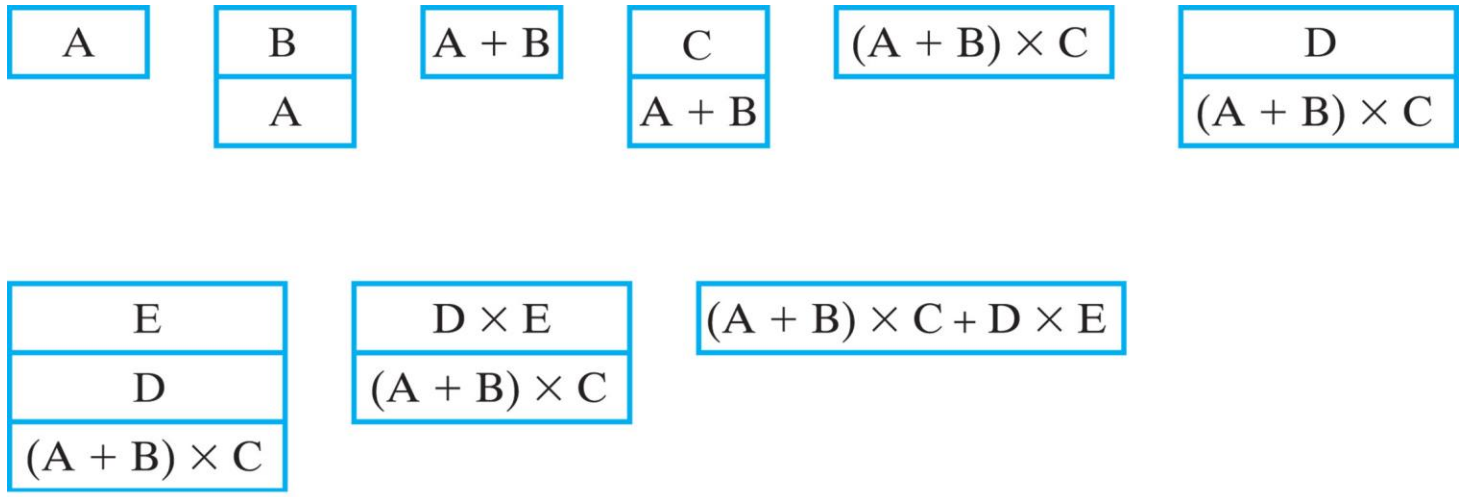
Addressing Architectures

- It is very easy to develop a program for an RPN expression.
- Whenever a variable is encountered, it is pushed onto the stack.
- Whenever an operation is encountered, it is executed on the implicit address TOS, or addresses TOS and TOS_{-1}, with the result placed in the new TOS.
- The program for the example RPN expression is

```
PUSH A
PUSH B
ADD
PUSH C
MUL
PUSH D
PUSH E
MUL
ADD
```

Addressing Architectures

- The execution of the program is illustrated by the successive stack states shown in Figure below.
- As an operand is pushed on the stack, the stack grows by one stack location.
- When an operation is performed, the operand in the TOS is popped off and temporarily stored in a register.
- The operation is applied to the stored operand and the new TOS operand, and the result replaces the TOS operand.



Addressing Modes

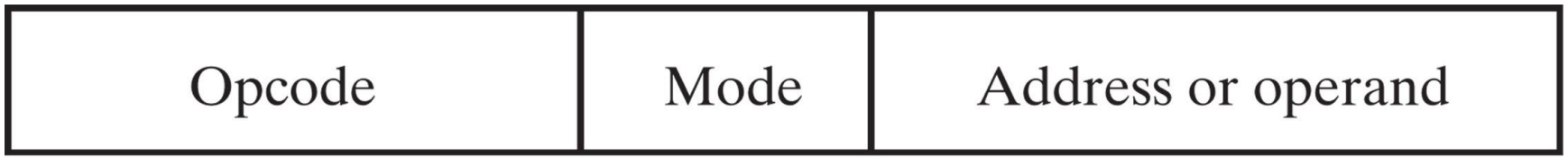
- The operation field of an instruction specifies the operation to be performed.
- This operation must be executed on data stored in computer registers or memory words.
- How the operands are selected during program execution is dependent on the addressing mode of the instruction.
- The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.
- The address of the operand produced by the application of such a rule is called the ***effective address***.
- Computers use addressing-mode techniques to accommodate one or both of the following provisions:
    1. To give programming flexibility to the user via pointers to memory, counters for loop control, indexing of data, and relocation of programs.
    2. To reduce the number of bits in the address fields of the instruction.

Addressing Modes

- The availability of various addressing modes gives the experienced programmer the ability to write programs that require fewer instructions.
- The effect, however, on throughput and execution time must be carefully weighed. For example, the presence of more complex addressing modes may actually result in lower throughput and longer execution time.
- Also, most machine-executable programs are produced by compilers that often do not use complex addressing modes effectively.
- In some computers, the addressing mode of the instruction is specified by a distinct binary code.
- Other computers use a common binary code that designates both the operation and the addressing mode of the instruction.
- Instructions may be defined with a variety of addressing modes, and sometimes two or more addressing modes are combined in one instruction.

Addressing Modes

- An example of an instruction format with a distinct addressing-mode field is shown in Figure below.
- The opcode specifies the operation to be performed.
- The mode field is used to locate the operands needed for the operation.
- There may or may not be an address field in the instruction.
- If there is, it may designate a memory address or a processor register.
- Moreover, as discussed in the previous section, the instruction may have more than one address field.
- In that case, each address field is associated with its own particular addressing mode.



Implied Mode

- Although most addressing modes modify the address field of the instruction, one mode needs no address field at all: **the implied mode**.
- In this mode, the operand is specified implicitly in the definition of the opcode.
- It is the implied mode that provides the location for the two-operand-plus-result operations when fewer than three addresses are contained in the instruction.
- For example, the instruction “complement accumulator” is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.
- In fact, any instruction that uses an accumulator without a second operand is an implied-mode instruction.
- For example, data-manipulation instructions in a stack computer, such as ADD, are implied-mode instructions, since the operands are implied to be on top of stack.

Immediate Mode

- In the immediate mode, the operand is specified in the instruction itself.
- In other words, an immediate-mode instruction has an operand field rather than an address field.
- The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction.
- Immediate-mode instructions are useful, for example, for initializing registers to a constant value.