Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Instruction Set Architecture

Schedule

- **Today and Wednesday**
  - Closer look at instruction sets
- **Fri**
  - **Quiz 4** (over Chapter 5, i.e. HW #11 and HW #12)

Endianness

- Many questions to answer when designing an instruction set:
  - Byte ordering (or **endianness**)?
    - If we have a two-byte integer, how is that stored in memory?

Endianness

- **What is a little endian computer system?**
  - Little-endian: lower bytes come first (stored in lower memory addresses)
  - Ex: Intel x86/x86-64
- **What is a big endian computer system?**
  - Higher bytes come first
  - Ex: IBM PowerPC

Gulliver's Travels

Endianness

- As an example, suppose we have the hexadecimal number `0x12345678`
  - i.e. bytes `0x12`, `0x34`, `0x56`, `0x78`
- The big endian and little endian arrangements of the bytes are shown below.

Lowest Address

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

Endianness

- **Seriously, why have two different ways to store data?**
- Big endian:
  - The sign of the number can always be determined by looking at the byte at address offset 0
  - Strings and integers are stored in the same order
- Little endian:
  - Makes it easier to place values on non-word boundaries.
  - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic
    - Take a 32-bit memory location with content 4A 00 00 00
    - Can read at the same address as either
      - 8-bit (value = 4A), 16-bit (004A), 24-bit (00004A), or 32-bit (0000004A),

Endianness

- Example: How is $19714C2F_{16}$ stored in little and big endian formats at address 140_{16}?
  - Little endian
    - $140_{16}=2F_{16}$
    - $141_{16}=4C_{16}$
    - $142_{16}=71_{16}$
    - $143_{16}=19_{16}$
  - Big endian
    - $140_{16}=19_{16}$
    - $141_{16}=71_{16}$
    - $142_{16}=4C_{16}$
    - $143_{16}=2F_{16}$

Endianness

- **How is DEADBEEF$_{16}$ stored in little and big endian formats at address 21C$_{16}$?**
  - Little endian
    - $21C_{16}=EF_{16}$
    - $21D_{16}=BE_{16}$
    - $21E_{16}=AD_{16}$
    - $21F_{16}=DE_{16}$
  - Big endian
    - $21C_{16}=DE_{16}$
    - $21D_{16}=AD_{16}$
    - $21E_{16}=BE_{16}$
    - $21F_{16}=EF_{16}$

Processor Data Storage

Instruction Formats

- Next design questions: How will the CPU store data?
- Three choices:
  1. A **stack** architecture
  2. An **accumulator** architecture
  3. A **general purpose register** architecture
- Tradeoffs
  - Simplicity (and cost) of hardware design
  - Execution speed
  - Ease of use

Stack vs Accumulator vs Register

- **Stack architecture**
  - Instructions and operands are implicitly taken from the stack
  - Stack cannot be accessed randomly
- **Accumulator architecture**
  - One operand of a binary operation is implicitly in the accumulator
  - One operand is in memory, creating lots of bus traffic
- **General purpose register (GPR) architecture**
  - Registers can be used instead of memory
  - Faster than accumulator architecture
  - Efficient implementation for compilers
  - Results in longer instructions

General Purpose Register Architectures

- Most systems today are GPR systems
- There are three types:
  - **Memory-memory** where two or three operands may be in memory
  - **Register-memory** where at least one operand must be in a register
  - **Load-store** where no operands may be in memory
- The number of operands and the number of available registers has a direct affect on instruction length

Stack Architecture

- Stack machines use one - and zero-operand instructions.
- LOAD and STORE instructions require a single memory address operand
- Other instructions use operands from the stack implicitly
- PUSH and POP operations involve only the stack's top element
- Binary instructions (e.g., ADD, MULT) use the top two items on the stack

Stack Architecture

- Stack architectures require us to think about arithmetic expressions a little differently
- We are accustomed to writing expressions using *infix notation*, such as: Z = X + Y
- Stack arithmetic requires that we use *postfix notation*: Z = XY+
  - This is also called **reverse Polish notation**, (somewhat) in honor of its Polish inventor, Jan Lukasiewicz (1878 – 1956)

Postfix Notation

- The principal advantage of postfix notation is that parentheses are not used
  - … plus it is easy to evaluate on a stack machine
- Infix expression
  - $Z = (X \times Y) + (W \times U)$
- Identical Postfix expression
  - $Z = X\ Y \times W\ U \times +$

Postfix Notation

- Example: Convert the infix expression to postfix
  - (2+3) - 6/3

2 3+ - 6/3	The sum 2 + 3 in parentheses takes precedence; we replace the term with 2 3 +.

Postfix Notation

- Example: Convert the infix expression to postfix
  - (2+3) - 6/3

2 3+ - 6 3/	The division operator takes next precedence; we replace 6/3 with 6 3 /.

Postfix Notation

- Example: Convert the infix expression to postfix
  - (2+3) - 6/3

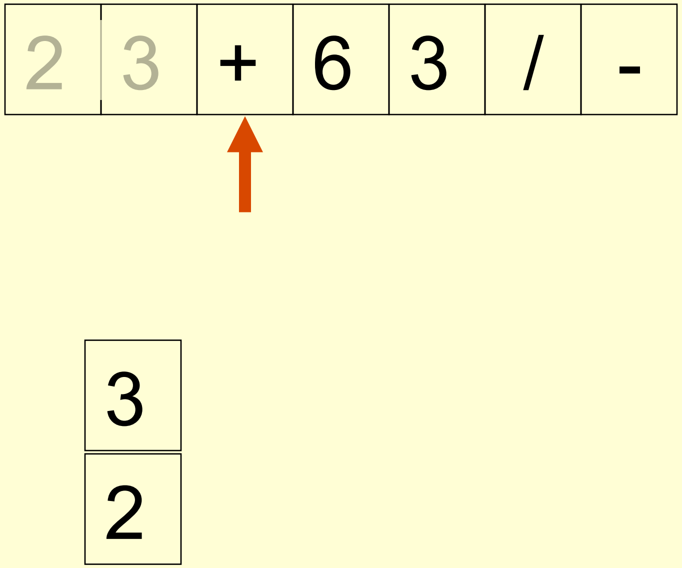
2 3+ 6 3/ -

The quotient 6/3 is subtracted from the sum of 2 + 3, so we move the - operator to the end.

Postfix Notation and Stacks

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / -

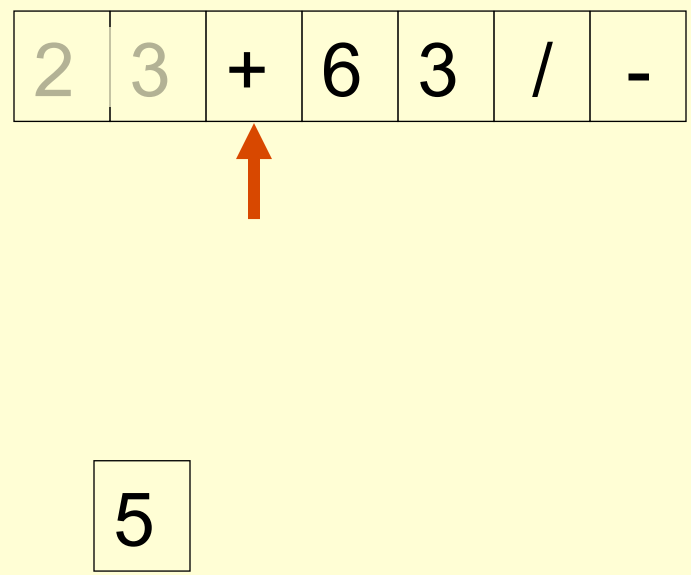
Scanning the expression from left to right, push operands onto the stack, until an operator is found



Postfix Notation and Stacks

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

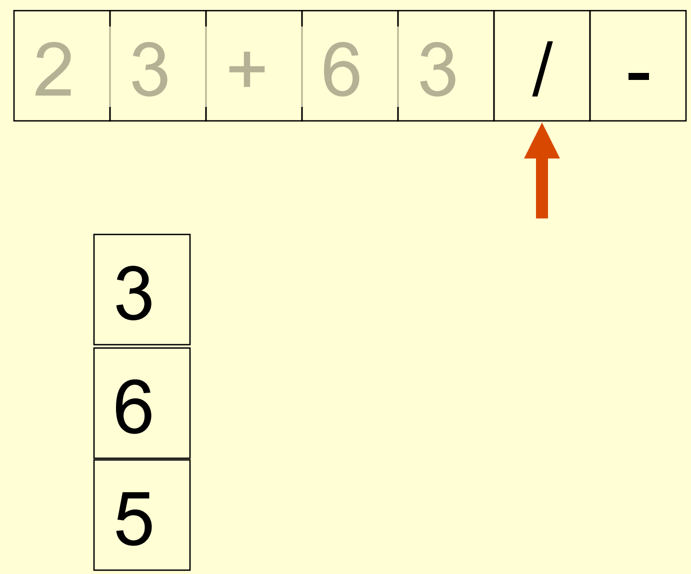
Pop the two operands and carry out the operation indicated by the operator. Push the result back on the stack.



Postfix Notation and Stacks

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

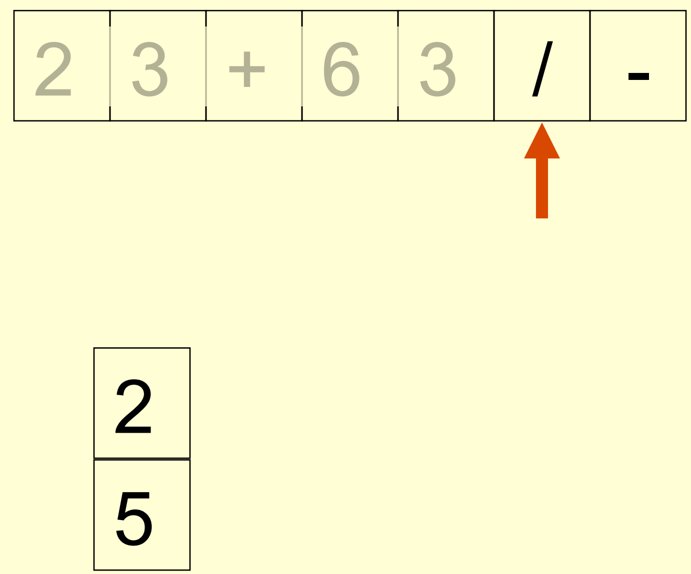
Push operands until another operator is found.



Postfix Notation and Stacks

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

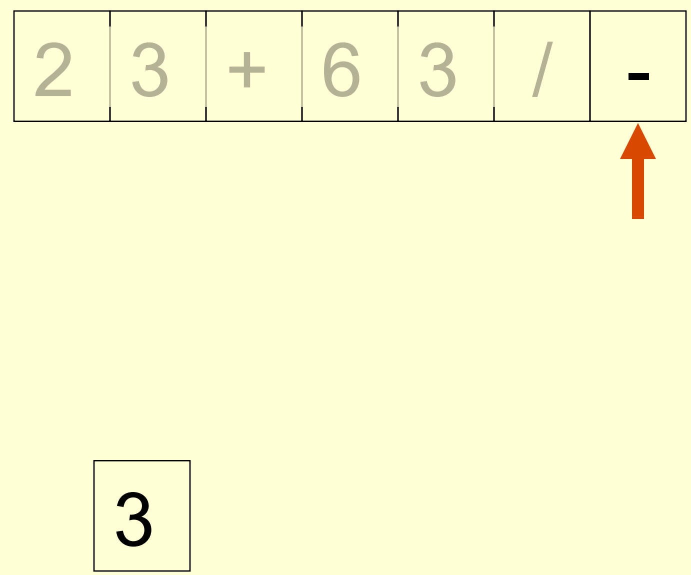
Carry out the operation and push the result.



Postfix Notation and Stacks

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Finding another operator, carry out the operation and push the result.
The answer is at the top of the stack.



Infix Expression and ISA

- Let's see how to evaluate an infix expression using different instruction formats
- With a three-address ISA, (e.g.,mainframes), the infix expression

  `Z = X × Y + W × U`

  might look like this

  - ```
    MULT R1,X,Y
    MULT R2,W,U
    ADD  Z,R1,R2
    ```
    ```

Infix Expression and ISA

- In a two-address ISA, (e.g., Intel, Motorola), the infix expression
  $Z = X \times Y + W \times U$
  might look like this
  - ```
    LOAD R1,X
    MULT R1,Y
    LOAD R2,W
    MULT R2,U
    ADD  R1,R2
    STORE Z,R1
    ```

Note: Two-address ISAs usually require one operand to be a register
    ```

Infix Expression and ISA

- In a one-address ISA, like MARIE, the infix expression `Z = X × Y + W × U` looks like this:
  - ```
    LOAD X
    MULT Y
    STORE TEMP
    LOAD W
    MULT U
    ADD TEMP
    STORE Z
    ```

Notice that as the instructions get shorter, the program gets longer…

Tradeoff – Hopefully these small instructions are faster than the large instructions!
    ```

Postfix Expression and ISA

- In a stack ISA, the postfix expression
  `Z = X Y × W U × +`
  might look like this:
  - ```
    PUSH X
    PUSH Y
    MULT
    PUSH W
    PUSH U
    MULT
    ADD
    POP Z
    ```

Would this program require more execution time than the corresponding (shorter) program that we saw in the 3-address ISA?
    ```

Postfix Expression and ISA

- **Implement the postfix expression**
  `Z = A B C + × D –`
  in a stack ISA

- **Convert the postfix expression to infix notation**

Postfix Expression and ISA

- **Implement the postfix expression**
  `Z = A B C + × D −`
  in a stack ISA
  - PUSH A
    PUSH B
    PUSH C
    ADD
    MULT
    PUSH D
    SUBT
    POP Z

- **Convert the postfix expression to infix notation**
  - Build up a stack to help convert back to infix notation
  - (A*(B+C)-D)

Instruction Types

Instruction types

- **7 broad categories** of processor instructions:
  - Data movement
  - Arithmetic
  - Boolean
  - Bit manipulation
  - I/O
  - Control transfer
  - Special purpose

Take 3 minutes and brainstorm examples of each

Instruction Types – Data Movement

- **Data movement**
  - Moves data between memory, registers, or both
- Examples
  - MARIE instructions: LOAD X and STORE X
  - PUSH and POP instructions
  - EXCHANGE: swap two values
  - May be different instructions for different sizes or types of data (LOADINT and LOADFLT)

Instruction Types - Arithmetic

- **Arithmetic**
  - Operations which involve the ALU to perform a calculation
- Examples
  - MARIE instructions: ADD X, SUBT X, ADDI X
  - MULTIPLY and DIVIDE
  - INCREMENT and DECREMENT: add or subtract 1 from a value
  - NEGATE: unary minus
  - Integer and floating point instructions
  - Some instruction sets even include scientific operations (SINE, SQRT, etc)

Instruction Types – Boolean

- **Boolean**
  - Logical operations on groups of bits
- Examples
  - AND X
    - Performs "bit-wise" operations

ACC	0	1	1	0	1	1	0	0
X	1	1	1	1	0	0	0	0
ACC	0	1	1	0	0	0	0	0

  - OR, NOT, XOR, COMPARE instructions

Instruction Types – Bit Manipulation

- **Bit manipulation**
  - Non-Boolean operations on bits
- Examples
  - ROTATE and SHIFT instructions
- ROTATE moves all bits left or right, and bits which are "shoved out" one side get "shoved in" the other
  - Example: ROTATEL 3 / rotate left 3 bits

ACC	0	1	0	0	0	0	1	1
ACC	0	0	0	1	1	0	1	0

Instruction Types – Bit Manipulation

- SHIFT moves all bits left or right, and bits which are "shoved out" are discarded
- For left shifts, 0's are shifted in
- For right shifts, the bits shifted in depends on whether the shift is logical or arithmetic
  - Logical: Shift in 0's
  - Arithmetic: Copy the leftmost bit (sign bit)
    - Thus, a negative number stays negative!

Instruction Types – I/O

- **Input / Output**
  - Transfer data from system to/from external devices
- Examples
  - MARIE instructions: INPUT and OUTPUT
  - Some processors have no special I/O instruction and instead use memory-mapped I/O, treating I/O devices like "special" memory

Instruction Types – Control Transfer

- **Control transfer**
  - Alter the normal sequence of program execution
- Examples
  - MARIE's JUMP, JUMPI, JNS, SKIPCOND, and HALT
  - Other processors have instructions like
    - BEQ/BNE (branch equal/not equal)
    - DJNZ (decrement and jump if not zero)
    - CJNE (compare and jump if not equal)

Instruction Types – Special Purpose

- Special purpose
  - Just about everything not covered above
  - These can provide access to special hardware specific to the CPU
    - Intel's SSE (Streaming SIMD Extensions) and AMD's 3DNow! instructions for multimedia applications
    - String manipulation instructions