Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

Instruction Set Architecture

Schedule

- **Today**
  - **Chapter 5** – Closer look at instruction sets
- **Next Tuesday**
  - *Continued…*
- **Next Thursday**
  - *Continued…*
  - Quiz 4

Today's Goals

- What factors are involved in instruction set architecture design?
- Look at different instruction formats, operand types, and memory access methods
  - A lot more possibilities than what MARIE offered in Chapter 4
- See the relationship between machine organization and instruction formats

Recap – Common Terms

- **Instruction Set Architecture (ISA)** - “Contract” between processor vendor and programmers
  - Instructions?
  - Registers?
  - Addressing modes?
  - Memory architecture?
  - Interrupt and exception handling?
  - I/O?
- **Opcode** – What instruction is being performed
- **Operand** – What data does does that instruction need?
  - Memory address, register name, etc…

Instruction Formats

- What makes instruction sets different?
  - Types of operations
  - Number of bits per instruction
  - Stack, accumulator, or register-based
  - Number of explicit operands per instruction
  - Operand location
  - Type and size of operands

Instruction Formats

- How can we measure different instruction set architectures? (in order to determine how "good" they are)
  - Main memory space occupied by a program
  - Instruction complexity
  - Instruction length (in bits)
  - Total number of instructions in the instruction set
- When designing an instruction set, you had better make the right decisions, since you'll be stuck with the architecture for decades! *(just ask Intel…)*

Instruction Formats

- Many questions to answer when designing an instruction set:
  - Instruction length?
    - Short? Long? Variable?
    - Shorter takes up less space in memory (good), but also reduces the number of possible instructions and the number of operands (bad)
    - Fixed length is easy to decode (good) but wastes space in memory (bad)

Instruction Formats

- Many questions to answer when designing an instruction set:
  - Number of operands?
  - Number of addressable registers?
  - Memory organization?
    - Whether byte- or word addressable
  - Addressing modes?
    - Choose any or all: **direct**, **indirect** or **indexed**

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:
  - Is more natural.
  - The sign of the number can 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 × Y) + (W × U)
- Identical Postfix expression
  - Z = X Y × W U × +

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

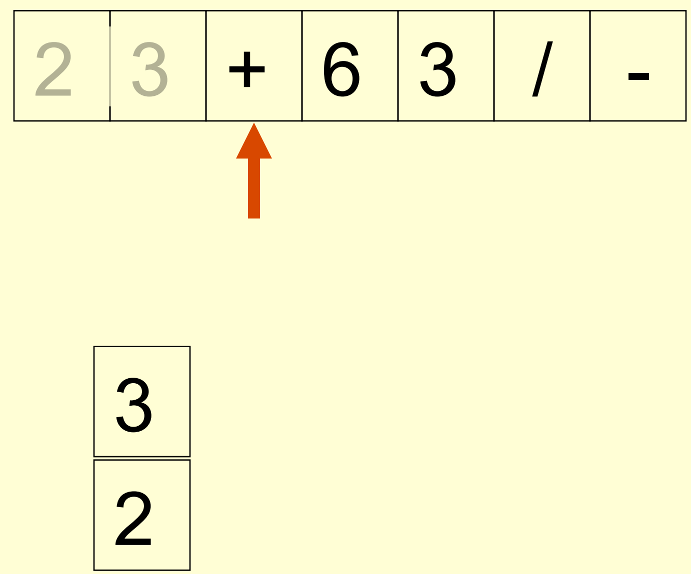
2 3+ 6 3/ -

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 / -

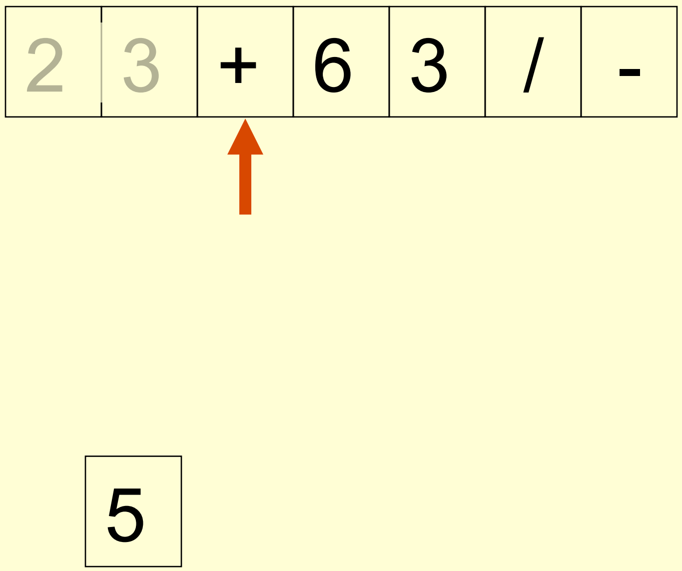
Scanning the expression from left to right, push operands onto the stack, until an operator is found



Postfix Notation and Stacks

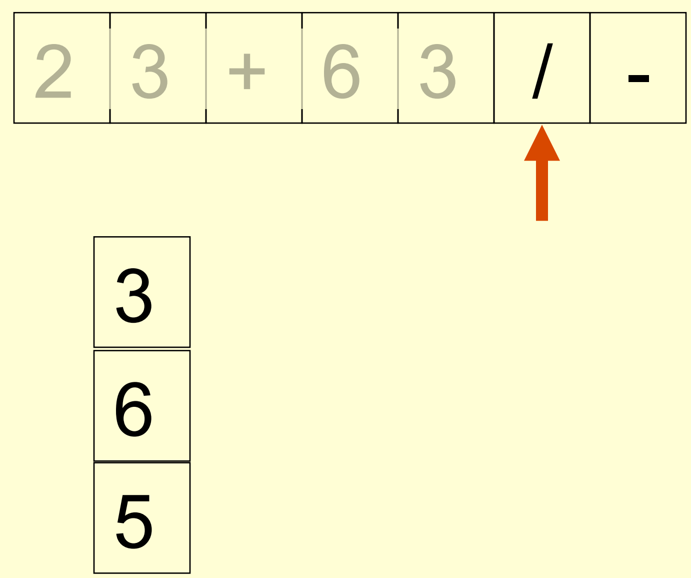
- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

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 / - :

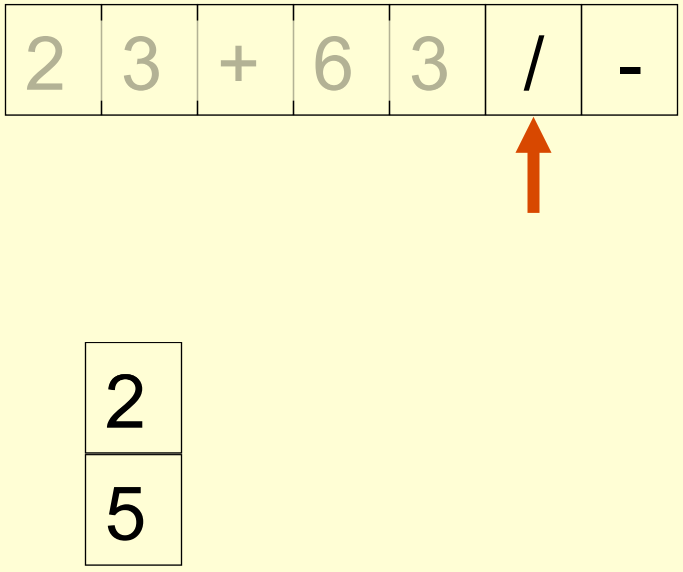


Push operands until another operator is found.

Postfix Notation and Stacks

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

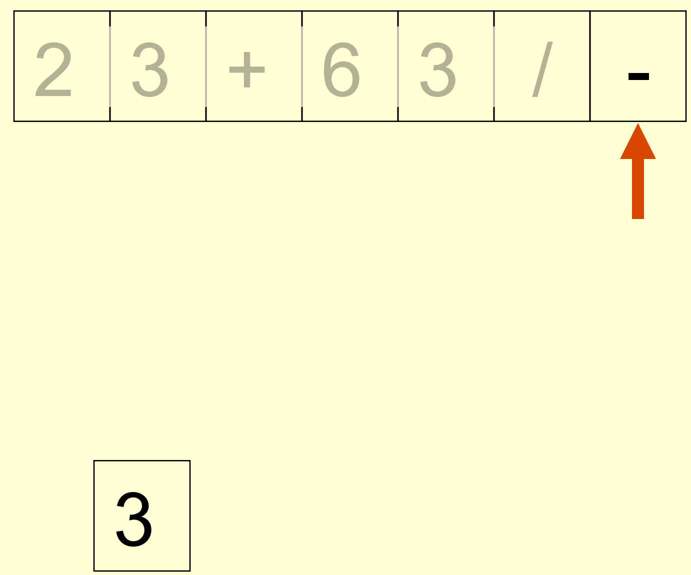
Carry out the operation and push the result.



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

  $Z = X \times Y + W \times U$

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

Expanding Opcodes

Expanding Opcodes

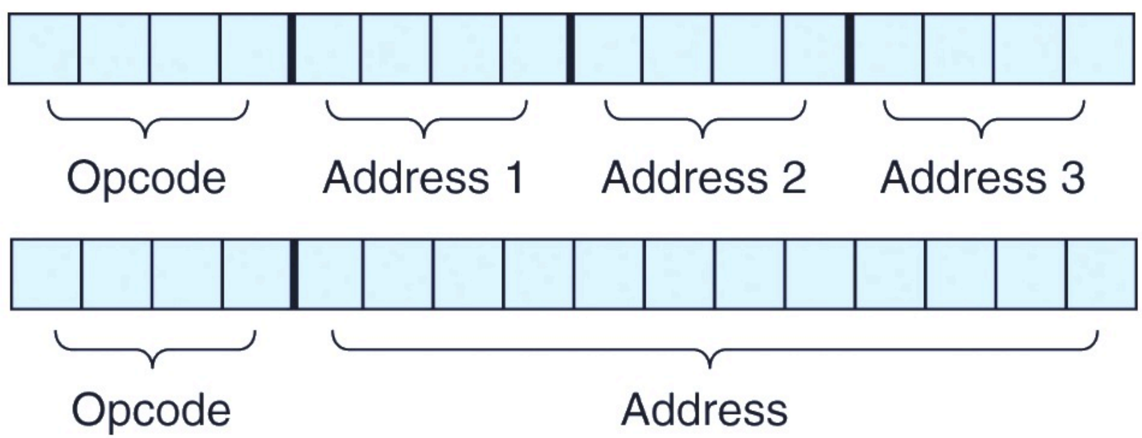
- We have seen how instruction length is affected by the number of operands supported by the ISA
- In any instruction set, not all instructions require the same number of operands
- Operations that require no operands, such as `HALT`, necessarily waste some space when fixed-length instructions are used
- One way to recover some of this space is to use *expanding opcodes*

Expanding Opcodes

- ISAs with expanding opcodes allow a varying number of opcode bits, depending on the needs of the instruction.
- The "trick" is to select opcode values so that certain bit patterns allow the opcodes to expand into what would be operand bits in other instructions

Expanding Opcodes

- A system has 16 registers and 4K of memory.
  - 4 bits is needed to access a register
  - 12 bits is needed for a memory address
- If the system is to have 16-bit instructions, we have two choices of instruction format:



Expanding Opcodes

- If we allow the length of the opcode to vary, we could create a very rich instruction set:

0000	R1	R2	R3	15 3-address codes
1110	R1	R2	R3	
1111 0000	R1	R2		14 2-address codes
1111 1101	R1	R2		
1111 1110 0000	R1			31 1-address codes
1111 1111 1110	R1			
1111 1111 1111 0000				16 0-address codes
1111 1111 1111 1111				

Expanding Opcodes

- A general expression for *this* machine, which gives the maximum number of possible opcodes of each type, is:

$$2^{16} = a * 2^{3*4} + b * 2^{2*4} + c * 2^{4} + d$$

where

- a = 3-register or 12-bit address opcodes,
- b = 2-register opcodes,
- c = 1-register opcodes, and
- d = 0-register opcodes

Expanding Opcodes

- Example: Given 8-bit instructions, is it possible to allow the following to be encoded?
  - 3 instructions with two 3-bit operands
  - 2 instructions with one 4-bit operand
  - 4 instructions with one 3-bit operand

We need:

$3 \times 2^3 \times 2^3$ = 192 bit patterns for the 2 3-bit operands

2×2^4 = 32 bits patterns for the 4-bit operands

4×2^3 = 32 bits patterns for the 3-bit operands

Total: 256 bits patterns (which equals 2^8)

Expanding Opcodes

- With a total of 256 bits required, we can exactly encode our instruction set in 8 bits!

We need:

3×2^3 = 192 bit patterns for the 3-bit operands

2×2^4 = 32 bits patterns for the 4-bit operands

4×2^3 = 32 bits patterns for the 3-bit operands.

Total: 256 bits patterns

One such encoding is shown on the next slide

Expanding Opcodes

```
00 xxx xxx
01 xxx xxx
10 xxx xxx
11 – escape opcode
1100 xxxx
1101 xxxx
1110 – escape opcode
1111 – escape opcode
11100 xxx
11101 xxx
11110 xxx
11111 xxx
```

- 00, 01, 10: 3 instructions with two 3-bit operands
- 1100, 1101: 2 instructions with one 4-bit operand
- 11100–11111: 4 instructions with one 3-bit operand

Expanding Opcodes

- The disadvantage of expanding opcodes is that it makes decoding logic more difficult
  - For a 3-address opcode, we only need to look at bits IR[15:12]
  - But for a 0-address opcode we (eventually) must look at all 16 bits of IR

Expanding Opcodes

- Example: Suppose we have a CPU with 12-bit long instructions, and 16 registers
  - We can have two, one and zero register instructions
- **If there are 13 two-register instructions and 39 one-register instructions, how many zero-register instructions can there be?**

Expanding Opcodes

- 16 registers required 4 bits to represent.
- In a 12-bit instruction, the opcode field is
  - 4 bits wide for two-register instructions
  - 8 bits wide for one-register instructions
  - 12 bits wide for zero-register instructions
- If there are 13 two-register instruction, there are 3 unused opcode combinations
  - This means there are 3*16 or 48 possible one-register instructions

Expanding Opcodes

- Since we have only 39 one-register instructions, there are again nine unused opcodes
  - This means there is room for 9*16 or **144 zero-address instructions**

$$n = 2^{12} - (13 * 2^{2*4} + 39 * 2^{4}) = 144$$

Expanding Opcodes

- EXERCISE
  - Suppose we have a CPU with 12-bit long instructions, and 16 registers
  - **If there are 13 two-register instructions and 256 zero-register instructions, how many one-register instructions can there be?**
    - 16 registers = 4 bits per register
    - Two-register instructions: 4 bits for opcode, 8 bits for operand

Expanding Opcodes

- 16 registers = 4 bits per register
- Two-register instructions
  - 4 bits for opcode, 8 bits for operand
  - 16 possible instructions, but we need 13
    - 3 opcodes remaining
    - 3 * 16 = 48 possible one-register instructions
- Zero-register instructions
  - We need 256 of these instructions
  - Last 4 bits unique (16 possibilities)
    - 256 / 16 = 16 (so need 16 options left in one-register)

Expanding Opcodes

- One-Register Instructions
  - 8 bits for opcode, 4 bits for operand
  - 48 possible 1-register instructions (see earlier)
  - But we need to save 16 as escape opcodes for the zero-register instructions
    - So, 48-16=**32 one-register instructions available**