CS 271
Computer Architecture & Assembly Language

Lecture 8

Error detection/correction

Modularization and MASM Procedures

1/27/21, Thursday

Oregon State University

Due Reminders

- Program #3
  - Due 1/30 11:59 PM on Canvas

- Weekly Summary Exercise 4
  - Due 1/30 11:59 PM on Canvas

Lecture Topics:

- Error-detecting and error-correcting codes
- Modularization
- MASM Procedures

Error-detecting and error-correcting codes

Simple Error Checking

- Parity is the total number of ‘1’ bits (including the extra parity bit) in a binary code
- Each computer architecture is designed to use either even parity or odd parity
- System adds a parity bit to make each code match the system’s parity

Parity (error checking)

- Example parity bits for 8-bit code 11010110
  - Even parity system: 111010110 (sets parity bit to 1 to make a total of 6 1-bits)
  - Odd parity system: 011010110 (sets parity bit to 1 to keep 5 1-bits)

- Code is checked for parity error whenever it is used.

- Examples for even-parity architecture:
  - 101010101 error (5 one-bits)
  - 100101010 OK (4 one-bits)
- Examples for odd-parity architecture:
  - 101010101 OK (5 one-bits)
  - 100101010 error (4 one-bits)

Parity (error checking)

- Used for checking memory, network transmissions, etc.
  - Error <u>detection</u>

- Not 100% reliable
  - Works only when error is in odd number of bits
  - … but very good because most errors are single-bit

A very short game

- For each of the following screens: [1, 31]
  - Write down the letter of the screen only if your birth date is on the screen

A

1	3	5	7
9	11	13	15
17	19	21	23
25	27	29	31

B

2	3	6	7
10	11	14	15
18	19	22	23
26	27	30	31

C

4	5	6	7
12	13	14	15
20	21	22	23
28	29	30	31

D

8	9	10	11
12	13	14	15
24	25	26	27
28	29	30	31

E

16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

A

1	3	5	7
9	11	13	15
17	19	21	23
25	27	29	31

B

2	3	6	7
10	11	14	15
18	19	22	23
26	27	30	31

C

4	5	6	7
12	13	14	15
20	21	22	23
28	29	30	31

D

8	9	10	11
12	13	14	15
24	25	26	27
28	29	30	31

E

16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

A

2^0

1 00001	3 00011	5 00101	7 00111
9 01001	11 01011	13 01101	15 01111
17 10001	19 10011	21 10101	23 10111
25 11001	27 11011	29 11101	31 11111

B

2^1

2 00010	3 00011	6 00110	7 00111
10 01010	11 01011	14 01110	15 01111
18 10010	19 10011	22 10110	23 10111
26 11010	27 11011	30 11110	31 11111

C

2^2

4 00100	5 00101	6 00110	7 00111
12 01100	13 01101	14 01110	15 01111
20 10100	21 10101	22 10110	23 10111
28 11100	29 11101	30 11110	31 11111

D

2^3

8 01000	9 01001	10 01010	11 01011
12 01100	13 01101	14 01110	15 01111
24 11000	25 11001	26 11010	27 11011
28 11100	29 11101	30 11110	31 11111

23 ABCE

$= 2^0 + 2^1 + 2^2 + 2^4$

$= 1 + 2 + 4 + 16$

E

$= 23$

16 10000	17 10001	18 10010	19 10011
20 10100	21 10101	22 10110	23 10111
24 11000	25 11001	26 11010	27 11011
28 11100	29 11101	30 11110	31 11111

2^4

Error-correcting: Hamming Codes

- n-bit code word ($n = m + r$)
  - m data bits
  - r check bits (to check parity)
  - There are 2^n possible code words
  - Only 2^m code words are valid
- Parity is the sum of one check bit and its selected data bits
  - May be even or odd
  - Used for detecting and correcting errors in memory, network transmissions, etc.
    - ECC memory, etc.

Parity check for single-bit errors

8 bits

- Number of parity bits depends on word size
  - Number of required parity bits (r) is $\log_2 m + 1$

$= \log_2 8 + 1 = 3 + 1 = 4$

- Guarantees Hamming distance of 2
  - i.e., to change one valid code to another valid code, at least 2 bits must be changed
  - If a valid code gets only one bit changed, the resulting code is invalid
- There are many invalid codes
  - Invalid codes indicate errors

Arranging the Parity Bits

- For 8 data bits, how many parity bits should be added? 4
- Number the bits left → right, 1→n
  - Note: different from usual numbering
- Bits numbered with powers of 2 are parity bits; others are data bits.

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
?	?		?				?				

2^0 2^1 2^2 2^3

Hamming code Example (p1)

- Represent decimal 45 as 8-bit with even parity Hamming code.
- m = 8, r = ($\log_2 8$ + 1) = 4, so n = 12
- 45 = 00101101 binary (8-bit)
- Fill in data bits, skipping the parity bits

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
?	?	0	?	0	1	0	?	1	1	0	1

Hamming code Example (p2)

- Parity bit #1 represents all place numbers having 1 in the 1's place (i.e., all odd-numbered places)
- Even parity requires that the count of '1' bits in these places (plus the parity bit) must be even.
  - There is one '1' data bit in these places, so set bit #1 to 1

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
1	?	0	?	0	1	0	?	1	1	0	1

Hamming code Example (p3)

- Parity bit #2 represents all place numbers having 1 in the 2's place
- There are two '1' data bits in these places, so set bit #2 to 0

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
1	0	0	?	0	1	0	?	1	1	0	1

Hamming code Example (p4)

- Parity bit #4 represents all place numbers having 1 in the 4's place
- There are two '1' data bits in these places, so set bit #4 to 0

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
1	0	0	0	0	1	0	?	1	1	0	1

Hamming code Example (p5)

- Parity bit #8 represents all place numbers having 1 in the 8's place
- There are three '1' data bits in these places, so set bit #8 to 1

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
1	0	0	0	0	1	0	1	1	1	0	1

Hamming code Example (p6)

- 45 = 00101101 (8-bit binary)
- 45 = 100001011101 (12-bit even parity Hamming Code)

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100
1	0	0	0	0	1	0	1	1	1	0	1

Hamming Code Error Example (p0)

- 100111110111 is a 12-bit odd-parity representation. Correct its single-bit error

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	1	1	1	1	1	0	1	1	1

Data bits are
01110111 = 119

Hamming Code Error Example (p1)

- 100111110111 is a 12-bit <u>odd-parity</u> representation. Correct its single-bit error

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	1	1	1	1	1	0	1	1	1

Data bits are
01110111 = 119

- 1s parity X

Hamming Code Error Example (p2)

- 100111110111 is a 12-bit odd-parity representation. Correct its single-bit error

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	1	1	1	1	1	0	1	1	1

Data bits are
01110111 = 119

- 1s parity X
- 2s parity X

Hamming Code Error Example (p3)

- 100111110111 is a 12-bit odd-parity representation. Correct its single-bit error

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	1	1	1	1	1	0	1	1	1

Data bits are
01110111 = 119

- 1s parity X
- 2s parity X
- 4s parity √

Hamming Code Error Example (p4)

- 100111110111 is a 12-bit odd-parity representation. Correct its single-bit error

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	1	1	1	1	1	0	1	1	1

Data bits are
01110111 = 119

- 1s parity X
- 2s parity X
- 4s parity √
- 8s parity X

Hamming Code Error Example (p5)

- 100111110111 is a 12-bit odd-parity representation. Correct its single-bit error
-

p	p	d	p	d	d	d	p	d	d	d	d
1	2	3	4	5	6	7	8	9	10	11	12
1	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**

Data bits are
01110111 = 119

- 1s parity X
- 2s parity X
- 4s parity √
- 8s parity X

The only bit that is in 1s and 2s and 8s and is NOT in 4s is bit number 11. Therefore, the number should be 100111110101.

The data bits should be 01110101 = 117

1+2+8 = 11

Internal Representation (Summary p1)

- Regardless of external representation, all I/O eventually is converted into electrical (binary) codes.
- Inside the computer, everything is represented by gates (open/closed)

Internal Representation (Summary p2)

- Since the number of gates in each group (byte, word, etc.) is finite, computers can represent numbers only within a finite range.

- Representations may be truncated; overflow/underflow can occur, and the Status Register will be set.

- Limited precision for floating-point representations.

$$\frac{a+b}{c} \overset{?}{=} \frac{a}{c} + \frac{b}{c}$$

✓ 20p ? 30p.

Internal Representation (Summary p3)

- Inside the computer
  - Bytes, words, etc., can represent a finite number of combinations of off/on switches.
  - Each distinct combination is called a code.
  - Each code can be used to represent:
    - Number value
    - Memory address
    - Machine instruction
    - Keyboard character
- **Representation is neutral**. The operating system and the programs decide how to interpret the codes.

Internal Representation

- You should be able to show the binary/hexadecimal representations of:
  - Integer values (signed / unsigned)
  - Characters (tables given on tests, don't memorize)
  - Floating-point values
  - Error-detecting codes (parity)
  - Error-correcting codes (Hamming)
- You should be able to convert representations
  - Binary ←→ Decimal
  - Decimal ←→ Hexadecimal
  - Hexadecimal ←→ Binary

Modularization and MASM Procedures

Modular Development (motivations)

- Team environment
  - Easier to divide the work and assign tasks
- Incremental testing
  - Easier to test “section” as you develop the program
- Reliability
  - Easier to verify (procedures/parameters)
- Debugging
  - Easier to find errors
- Maintenance
  - Easier to make changes
- Re-usable code
  - Easier to use library modules (don’t re-invent)

MASM Procedures

- The “sections” of your main program are like modules (for modular development)
- Convert the “modules” into procedures
- Convert the main procedure into the control module
  - The control module calls the procedures
    - main should be (mostly) procedure calls
  - Any procedure may call another procedure
    - … or call itself (recursion: more later)

Program Design Using Procedures

- Top-Down Design (functional decomposition) involves the following:
  - Design your program before starting to code
  - Separate the program into its major tasks
  - Break large tasks into smaller ones
  - Use a hierarchical structure based on procedure calls
  - Test individual procedures separately

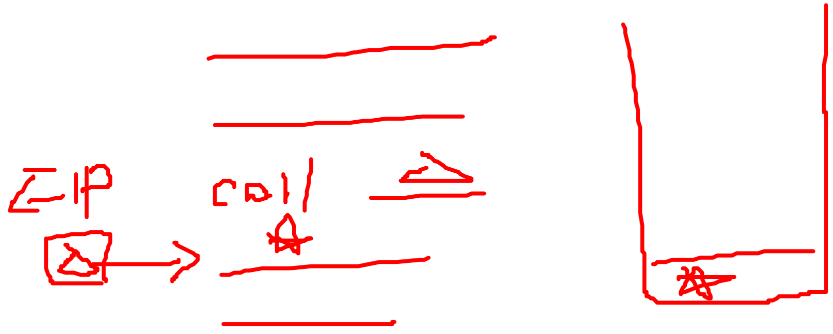
Creating Procedures

- A procedure is the assembly equivalent of a Java method or C/C++ function.
- Example procedure named *sample*:

```
sample PROC
   .
   .
   ret
sample ENDP
```

- Note that it looks a lot like *main*
  - Has **`ret`** instead of **`exit`**

CALL and RET Instructions



- The CALL instruction calls a procedure
  1. Pushes the offset of the next instruction in the calling procedure onto the system stack
     - i.e. pushes the current address in the EIP register onto the system stack (recall the Instruction Execution Cycle?
  2. Copies the address of the called procedure into EIP
  3. Executes the called procedure until RET
- The RET instruction returns control to the calling procedure
  - Pops the top of stack into EIP
    - i.e., execution continues in the calling procedure at the instruction after the CALL

- Note: Much more later about the system stack

Programming with Procedures

1. Write main to decide what variables and procedures are needed
2. Write procedure stubs
   - test
3. Declare variables
   - test
4. Implement procedures one at a time
   - Test each procedure separately
   - Recommended order of implementation:
     - Output, input, process
5. Do the housekeeping (document procedures)
   - test

Example program:
Using Top-Down Design

- Get 2 integers (a and b) from the user
- Find the summation of integers in [a…b]
- Display the result