CSED211 컴퓨터SW시스템개론

Homwork #2

20180551
컴퓨터공학과
이준석

1. Investigate the following things.

A. What is 'Instruction Set Architecture' (ISA)?

- 마이크로프로세서가 인식해서 기능을 이해하고 실행할 수 있는 기계어 명령어를 말한다. 마이크로프로세서마다 기계어 코드의 길이와 숫자 코드가 다르다. 명령어의 각 비트는 기능적으로 분할하여 의미를 부여하고 숫자화한다. 프로그램 개발자가 숫자로 프로그램하기가 불편하므로 기계어와 일대일로 문자화한 것이 어셈블리어이다.
- ISA는 하드웨어와 소프트웨어 사이의 인터페이스 역할을 한다.
- ISA는 기계어 프로그래밍을 위한 모든 것을 정의한다. ISA가 정의하는 것은 ISA의 종류마다 다르지만, 일반적으로 data type, state, semantics, instruction set, input/output model을 정의한다.

B. What is 'RISC' architecture? What are the big difference compared to 'CISC'?

- RISC란 Reduced Instruction Set Computer의 줄임말이다. 이름이 가지는 의미 그대로 CPU 명령어의 개수를 줄여 하드웨어 구조를 좀 더 간단하게 만드는 방식으로, 마이크로프로세서를 설계하는 방법 가운데 하나이며, SPARC, MIPS 등의 아키텍처에서 사용된다.
- RISC의 특징

  1. 고정 길이의 명령어를 사용하여 더욱 빠르게 해석할 수 있다.
  2. 모든 연산은 하나의 클럭으로 실행되므로 파이프라인을 기다리게 하지 않는다.
  3. 레지스터 사이의 연산만 실행하며, 메모리 접근은 로드(load), 스토어(store) 명령어로 제 한된다. 이렇게 함으로써 회로가 단순해지고, 불필요한 메모리 접근을 줄일 수 있다.
  4. 마이크로코드 논리를 사용하지 않아 높은 클럭을 유지할 수 있다.
  5. 많은 수의 레지스터를 사용하여 메모리 접근을 줄인다.
  6. 지연 실행 기법을 사용하여 파이프라인의 위험을 줄인다.

- RISC와 CISC의 가장 큰 차이점은 바로 명령어의 개수라고 할 수 있다. RISC가 더 적은 수의 명령어를 갖는다. 아래의 도표는 CISC와 RISC를 대조하여 나타낸 도표이다.

CISC	RISC
The original microprocessor ISA	Redesigned ISA that emerged in the early 1980s
Instructions can take several clock cycles	Single-cycle instructions
Hardware-centric design – the ISA does as much as possible using hardware circuitry	Software-centric design – High-level compilers take on most of the burden of coding many software steps from the programmer
More efficient use of RAM than RISC	Heavy use of RAM (can cause bottlenecks if RAM is limited)
Complex and variable length instructions	Simple, standardized instructions
May support microcode (micro-programming where instructions are treated like small programs)	Only one layer of instructions
Large number of instructions	Small number of fixed-length instructions
Compound addressing modes	Limited addressing modes

https://www.microcontrollertips.com/risc-vs-cisc-architectures-one-better/

2. Read carefully section 3.6.6 and answer the following questions

A. Why is the conditional move advantageous?

mis-predicted branch의 비용은 매우 크다. 여러 번의 cycle을 희생해야 한다. 하지만 conditional move는 prediction을 사용하지 않고 그렇기에 mis-predict로 인해 발생하는 cycle의 손해라는 단점이 존재하지 않는다. 그렇기에 conditonal move가 이득이 되는 부분이 있는 것이다.

B. As an invalid use of conditional move, it has shown an example of the following code (page 254, 3nd ed.)

```
long cread (long *xp) {
 return ( xp ? *xp : 0 );
} // if xp is not null pointer, then return the value by dereferencing
// if a compiler generates an assembly code like below, it may make a problem during
runtime. (Definitely, gcc compiler will not generate an assembly code like below.)
cread:
movq (%rdi), %rax
testq %rdi, %rdi
movl $0, %edx
cmove %rdx, %rax
ret
```

Explain what is wrong in this example.

conditional move는 조건과 무관하게 then-expr과 else-expr을 모두 실행하고 둘 중에 어떤 것을 선택할지 나중에 결정한다. 그렇기에 위의 삼항 연산자 ? : 에서 *xp와 0을 모두 실행하게 되는데, 여기서 문제가 되는 것은 *xp 부분이다. 만약에 xp가 null pointer 였다면 아무것도 가리키지 않는 포인터를 참조하려고 시도(movq (%rdi), %rax)했으므로 오류가 발생할 것이다.

3. Exercise 3.60 on page 348

3.60 ◆◆
Consider the following assembly code:

```
    long loop(long x, int n)
    x in %rdi, n in %esi
1   loop:
2     movl    %esi, %ecx
3     movl    $1, %edx
4     movl    $0, %eax
5     jmp     .L2
6   .L3:
7     movq    %rdi, %r8
8     andq    %rdx, %r8
9     orq     %r8, %rax
10    salq    %cl, %rdx
11  .L2:
12    testq   %rdx, %rdx
13    jne     .L3
14    rep; ret
```

The preceding code was generated by compiling C code that had the following overall form:

```
1   long loop(long x, long n)
2   {
3       long result = _____;
4       long mask;
5       for (mask = _____; mask  _____ ; mask = _____ ) {
6           result |= _____ ;
7       }
8       return result;
9   }
```

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register `%rax`. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables.

A. Which registers hold program values `x`, `n`, `result`, and `mask`?
B. What are the initial values of `result` and `mask`?
C. What is the test condition for `mask`?
D. How does `mask` get updated?
E. How does `result` get updated?
F. Fill in all the missing parts of the C code.

[A] rdi에 x, rsi에 n, rax에 result, rdx에 mask
[B] result의 initial value는 0, mask의 initial value는 1
[C] mask != 0
[D] mask = mask << n
[E] result |= (x & mask)

[F]

```
long result = 0;
long mask;
for( mask = 1; mask != 0; mask = mask << n ) {
        result |= (x & mask);
}
```

4. Exercise 3.63 on page 350

3.63 ◆◆
This problem will give you a chance to reverse engineer a `switch` statement from disassembled machine code. In the following procedure, the body of the `switch` statement has been omitted:

```
1  long switch_prob(long x, long n) {
2      long result = x;
3      switch(n) {
4          /* Fill in code here */
5
6      }
7      return result;
8  }
```

Figure 3.53 shows the disassembled machine code for the procedure.

The jump table resides in a different area of memory. We can see from the indirect jump on line 5 that the jump table begins at address `0x4006f8`. Using the GDB debugger, we can examine the six 8-byte words of memory comprising the jump table with the command `x/6gx 0x4006f8`. GDB prints the following:

```
(gdb) x/6gx 0x4006f8
0x4006f8:    0x00000000004005a1    0x00000000004005c3
0x400708:    0x00000000004005a1    0x00000000004005aa
0x400718:    0x00000000004005b2    0x00000000004005bf
```

Fill in the body of the `switch` statement with C code that will have the same behavior as the machine code.

```
     long switch_prob(long x, long n)
     x in %rdi, n in %rsi
1    0000000000400590 <switch_prob>:
2      400590:  48 83 ee 3c             sub    $0x3c,%rsi
3      400594:  48 83 fe 05             cmp    $0x5,%rsi
4      400598:  77 29                   ja     4005c3 <switch_prob+0x33>
5      40059a:  ff 24 f5 f8 06 40 00    jmpq   *0x4006f8(,%rsi,8)
6      4005a1:  48 8d 04 fd 00 00 00    lea    0x0(,%rdi,8),%rax
7      4005a8:  00
8      4005a9:  c3                      retq
9      4005aa:  48 89 f8                mov    %rdi,%rax
10     4005ad:  48 c1 f8 03             sar    $0x3,%rax
11     4005b1:  c3                      retq
12     4005b2:  48 89 f8                mov    %rdi,%rax
13     4005b5:  48 c1 e0 04             shl    $0x4,%rax
14     4005b9:  48 29 f8                sub    %rdi,%rax
15     4005bc:  48 89 c7                mov    %rax,%rdi
16     4005bf:  48 0f af ff             imul   %rdi,%rdi
17     4005c3:  48 8d 47 4b             lea    0x4b(%rdi),%rax
18     4005c7:  c3                      retq
```

Figure 3.53 Disassembled code for Problem 3.63.

```
long switch_prob(long x, long n) {
        long result = x;
        switch(n) {
        case 60:
        case 62:
                result = 8 * x;
                break;
        case 63:
                result = x;
                result = result >> 3;
                break;
        case 64:
                result = x;
                result = result << 4;
                result -= x;
                x = result;
                x = x*x;
                result = x + 16 * 4 + 11;
                break;
        case 65:
                x = x*x;
                result = x + 16 * 4 + 11;
                break;
        default:
                result = x + 16 * 4 + 11;
                break;
        }
        return result;
}
```

5. Exercise 3.66 on page 353

```
1  long sum_col(long n, long A[NR(n)][NC(n)], long j) {
2      long i;
3      long result = 0;
4      for (i = 0; i < NR(n); i++)
5          result += A[i][j];
6      return result;
7  }
```

In compiling this program, GCC generates the following assembly code:

```
   long sum_col(long n, long A[NR(n)][NC(n)], long j)
   n in %rdi, A in %rsi, j in %rdx
1  sum_col:
2    leaq    1(,%rdi,4), %r8
3    leaq    (%rdi,%rdi,2), %rax
4    movq    %rax, %rdi
5    testq   %rax, %rax
6    jle     .L4
7    salq    $3, %r8
8    leaq    (%rsi,%rdx,8), %rcx
9    movl    $0, %eax
10   movl    $0, %edx
11 .L3:
12   addq    (%rcx), %rax
13   addq    $1, %rdx
14   addq    %r8, %rcx
15   cmpq    %rdi, %rdx
16   jne     .L3
17   rep; ret
18 .L4:
19   movl    $0, %eax
20   ret
```

Use your reverse engineering skills to determine the definitions of NR and NC.

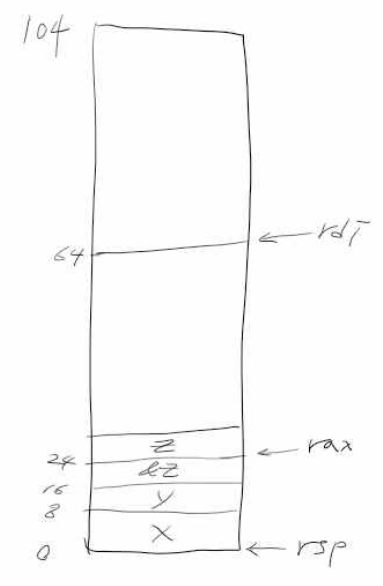
NR은 for문에서 test 조건을 확인하면 그 크기를 알 수 있을 것이다.
15번째 줄의 cmpq가 있다. rdi와 rdx(rdx는 i이다.)의 값을 비교하는데,
여기서 rdi는 3*n(3, 4번 째 줄에 의해서)이다. 따라서 NR은 3 * n 이다.

NC는 for문 안에서 다음 행으로 이동할 때 몇 칸 씩 이동하는지 확인해보면 열의 크기를 확인할 수 있으므로 NC도 알 수 있다. 14번 째 줄에서 r8 만큼 이동하는 것을 알 수 있다. r8은 4 * rdi + 1 이므로 NC의 값은 4 * n + 1 이다.

NR : 3 * n
NC : 4 * n + 1

6. Exercise 3.67 on page 354

A. We can see on line 2 of function eval that it allocates 104 bytes on the stack. Diagram the stack frame for eval, showing the values that it stores on the stack prior to calling process.



B. What value does eval pass in its call to process?

rdi에 (rsp + 64)의 주소를 저장하여 pass한다.

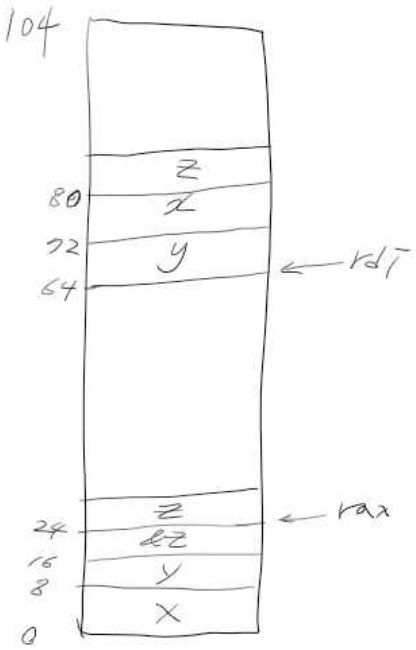
C. How does the code for process access the elements of structure argument s?

eval에서 process를 호출할 때 return address가 stack에 push된다. 그리고 rsp를 바탕으로 z, y, x에 접근하여 rdi가 가리키고 있는 부분 쪽에 값을 저장한다.

D. How does the code for process set the fields of result structure r?

처음에 pass된 rdi((rsp+64)의 주소)를 이용하여 r에 값을 저장(set)한다.

E. Complete your diagram of the stack frame for eval, showing how eval accesses the elements of structure r following the return from process.



process 함수가 완료된 후, rsp는 다시 주소 '0'을 가리킨다. 그래서 (rsp + 72의 값) + (rsp + 64의 값) + (rsp + 80의 값)을 rax에 저장하여 최종적인 연산을 완료한다.

F. What general principles can you discern about how structure values are passed as function arguments and how they are returned as function results?

구조체가 함수의 인자로서 전달될 때, 구조체의 값들은 스택에 저장되고 다루어진다.
구조체를 반환하는 함수가 호출될 때, caller는 스택에 공간을 할당하고 그 할당한 공간의 주소를 callee에게 pass한다.

7. Exercise 3.69 on page 357

3.69 ◆◆◆
You are charged with maintaining a large C program, and you come across the following code:

```
1  typedef struct {
2      int first;
3      a_struct a[CNT];
4      int last;
5  } b_struct;
6
7  void test(long i, b_struct *bp)
8  {
9      int n = bp->first + bp->last;
10     a_struct *ap = &bp->a[i];
11     ap->x[ap->idx] = n;
12 }
```

The declarations of the compile-time constant CNT and the structure a_struct are in a file for which you do not have the necessary access privilege. Fortunately, you have a copy of the .o version of code, which you are able to disassemble with the OBJDUMP program, yielding the following disassembly:

```
     void test(long i, b_struct *bp)
     i in %rdi, bp in %rsi
1    0000000000000000 <test>:
2       0:   8b 8e 20 01 00 00       mov    0x120(%rsi),%ecx
3       6:   03 0e                   add    (%rsi),%ecx
4       8:   48 8d 04 bf             lea    (%rdi,%rdi,4),%rax
5       c:   48 8d 04 c6             lea    (%rsi,%rax,8),%rax
6      10:   48 8b 50 08             mov    0x8(%rax),%rdx
7      14:   48 63 c9                movslq %ecx,%rcx
8      17:   48 89 4c d0 10          mov    %rcx,0x10(%rax,%rdx,8)
9      1c:   c3                      retq
```

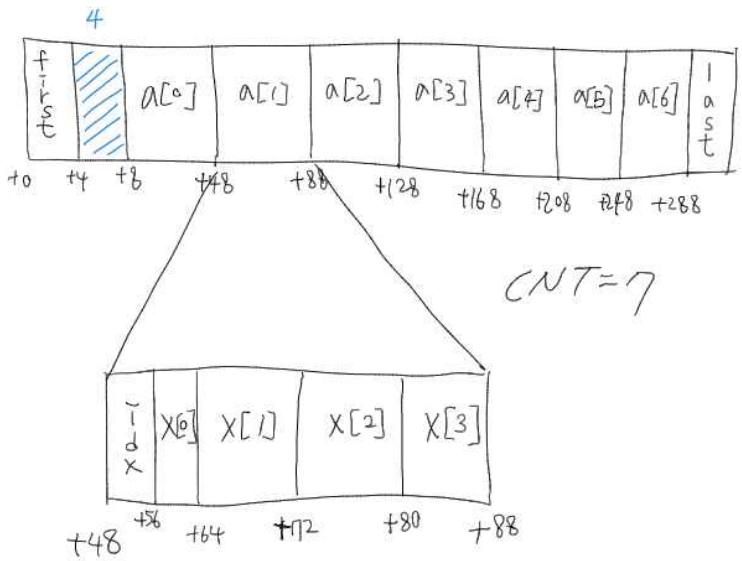


그림 1

- Aligned Data
  - Primitive data type requires ***B*** bytes
  - Address must be multiple of ***B***

그림 2

그림 1은 aligned data에 관한 규칙에 따라, 어셈블리 코드를 따라가며 구조체의 그림을 그려본 것이다. 이것을 가지고 문제를 풀어본다.

A. The value of CNT.

CNT = 7

B. A complete declaration of structure a_struct. Assume that the only fields in this structure are idx and x, and that both of these contain signed values.

```
typedef struct {
        int idx;
        int x[4];
} a_struct;
```