Machine-level Programming (4)

Assembler Directives

- Not instructions but help assembler and linker do their jobs
  - Reserve data locations
  - Sectioning
  - etc.

Components of assembly language (1/2)

- Reserved words
  - Instruction: ex) move, add, jmp
  - Directive: ex) .title, .list, .word
  - Operator: ex) +, −, *, /, &&, ||
- Identifiers
  - Type
    - name: indicate address of data
    - label: indicate address of instruction
  - Available character: A – Z, a – z, 0 – 9, ., _, $
  - Available first character: A – Z, a – z, ., _, $
  - Length: Unlimited. But, special symbols(new line, carriage return, etc.)are not allowed.

Components of assembly language (2/2)

- Statement
  - Type
    - Instruction: converted to object code
    - Directive: make assembler take specific action
  - Format

    label   operation   operand   ;comment

    - All components are selective.
  - 예) iadd: addl %eax, $4(%edx)   ; add integers
- Comment
  - Start with a semicolon(;)
  - Location: if the first field begins with a semicolon, the whole line comment or comment field of the sentence.

Components of assembly language

Label	Opcode	Operands	Comments
byte_num:			; Only label.
	.byte	0xa1, 0x89	; Directive – reserve byte.
	Movl	%eax, %ebx	; Instruction – move data.
label5:	incw	%cx	; Instruction – increase cx by 1.
	Nop		; Only instruction. No OPeration.
.L1:			; Only label.
	addl	$32, %esp	; Instruction – esp ← 0x20
	popl	%esi	; Instruction – esi ← (esp)
	popl	%edi	; Instruction – edi ← (esp)
	leave		; Instruction – restore stack for
			; procedure exit.

```
leave = (movl %ebp, %esp | popl %ebp)
```

Directives

- Directives
  - Command processed by assembler when assembly program is assembled.
  - Therefore, those are not translated to machine language instruction.
- Example of directives
  - Data allocation and definition directives
    - .byte expressions, .word exp, .long exp (= .int exp)
    - 예) num_w: .word 0x438a, 0x439a, 0x43aa ; *num_w* is a label
      .ascii “Get smart!” ; a string constant
  - Directives which help assembler and linker without machine language translation.
    - .section
      - .section name[, “flags”]
      - Assemble code that follows this directive to section “name”.
      - Flags indicate whether those are available, writable or executable in the ELF format.
      - ex) .section data, “w” ; *writable* section name = *data*
    - .text, .data
    - .align, .p2align
  - Directives for comment
    - .file
    - .ident

Directives

Directive	인자	Meaning	Example
.file	*Filename*	Specify begin of new file (only located in begin of assembly file)	.file “new_c”
.align	*boundary, fill_val, max_n*	Align next line with address which is a multiple of “boundary”(in case of a.out, boundary=number of 0 of binary). Fill blanks between those in “fill_val” up to “max_n”.	.align 8 (ELF) = .align 3 (a.out)[1]
.p2align	*boundary, fill_val, max_n*	Same with .align. But value of “boundary”[like case of a.out] is the number of 0 of binary.	.p2align 3, 0xff
.text	*Subsection*	Allocate subsequent sentence in text “subsection”	.text 0 ; .text 1
.data	*Subsection*	Allocate subsequent sentence in data “subsection”	.data 0 ; data 1
.globl	*Symbol*	Expose symbol to linker “ld”	.globl (=global) func5
.set	*symbol, exp*	Set “symbol value to “exp”	.set ten, 10
.ident	*“string”*	Simply, record “string” in comment section of the object file	.ident "GCC: (GNU) 3"
.org	*new_p, fill_val*	Change Location Counter to “new_p” in same section. Fill blanks between those with “fill_val”	.org next_pos
.include	*Filename*	Include “filename”	.include “infile.h”

1: Specify format of object and executable files in ELF (executable and linking format). Replace initial a.out format.

Directives

- About “section”
  - Section is a field of address space which is consisted of adjacent addresses.
  - When linker(“id”) links many object files to a executable, it does in section units.
- Type of section – GAS and linker(ld)
  - text section, data section
    - Accept programs. Distinct section but treated as same kind.
    - Text section cannot be edited in runtime. Mostly, It is shared between processes.
    - Data section can be edited in runtime. For example, variables of C is saved in it.
  - bss section (bss: blow stack segment)
    - Accept variables or shared memory space. 0 in startup execution.
- subsection
  - Within each section, subsection can be placed.
  - Identification number of each subsection is from 0 to 8192.
  - Object of same subsection is adjacently(in the same place) placed during generating object file.

Mixed-mode programming

Mixed-mode programs

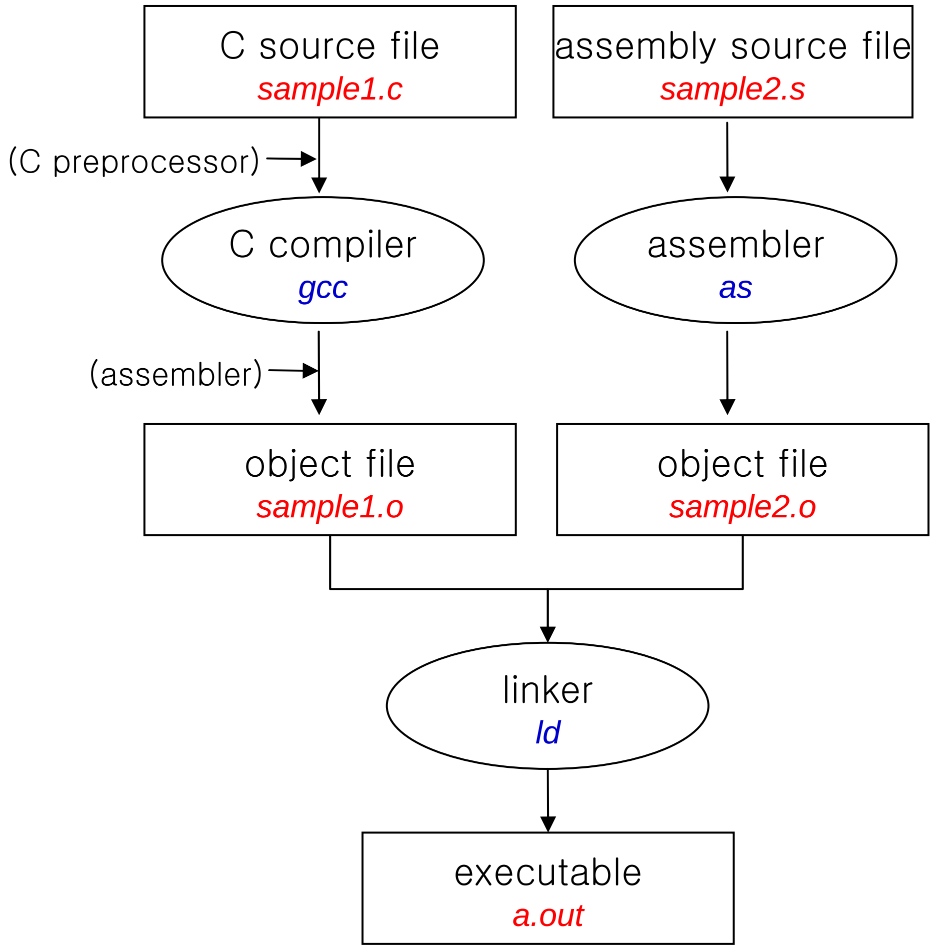
- Mixed-mode programs
  - Program which is written using high level language such as C, C++ and Pascal and low level language such as assembly language.
- Pros and cons of assembly language programming
  - Pros
    - Easy access to hardware
    - Time and space efficiency
  - Cons
    - Lower productivity of program development. Higher maintenance cost.
    - Absence of portability. Codes developed in an architecture cannot be used in different environment.
- Purpose of mixed-mode programs
  - Device driver programming
    - When direct access to hardware using common C language is impossible.
  - Optimization of program code to make faster execution speed
  - Use extended CPU instruction set
    - Unavailable instruction set in common C language
    - 예) Intel MMX extended multimedia instruction set
  - Receive API type system service in assembly program

Mixed-mode programs

- Rule for mixed-mode programs
  - Need
    - It should be the common rule between codes because mixed-mode programs are made by integrating many program module written with different languages or plural languages are used in a program.
  - Range of rule
    - Shared identifier symbol
      - For example, add "_"(underscore) in front of assembly processor called by C program.
    - Calling convention
      - Two languages should follow same calling convention.
      - Argument passing procedure: for example, pass from right argument using stack.
      - Return value handling
    - resister handling
      - Properly preserve value of resisters depending on caller-save or callee-save.
- How to write mixed-mode programs
  - Write program modules in different languages and integrate.
    - ex) Call processor written in 80x86 assembly language in a program written in C language. Or, in contrast, call C processor in assembly program.
  - Use of inline assembly
    - Insert assembly code directly in high level language program

Mixed-mode programming

- Procedure of mixed-mode programming
  - According to given rules, write high level language program and low level language program.
  - After compile/assemble each of them, generate executable by linking.
  - Example of procedure:



Mixed-mode programming

- Call assembly program in C program

```
1> gcc  –O  –c  code_c3_main.c  –o main.o
```

```
extern int proc_2(int x, int y);

void main()
{
		int x=10, y=20, s;
		s = proc_2(x,y);
}
```

```
2> as  code_c3_proc2.s  –o  proc_2.o
```

```
		.globl proc_2
		.globl _start
		.type		proc_2, @function
proc_2:
_start:
		pushl		%ebp
		movl		%esp, %ebp
		movl		12(%ebp), %eax
		addl		8(%ebp), %eax
		leave
		ret
```

```
3> ld  main.o, proc_2.o
4> objdump  –d  a.out  >main.dis
```

```
a.out:     file format elf32-i386

Disassembly of section .text:

08048094 <main>:
 8048094:   55                      push   %ebp
 8048095:   89 e5                   mov    %esp,%ebp
 8048097:   83 ec 08                sub    $0x8,%esp
 804809a:   83 e4 f0                and    $0xfffffff0,%esp
 804809d:   83 ec 18                sub    $0x18,%esp
 80480a0:   6a 14                   push   $0x14
 80480a2:   6a 0a                   push   $0xa
 80480a4:   e8 03 00 00 00          call   80480ac <proc_2>
 80480a9:   c9                      leave
 80480aa:   c3                      ret
 80480ab:   90                      nop

080480ac <proc_2>:
 80480ac:   55                      push   %ebp
 80480ad:   89 e5                   mov    %esp,%ebp
 80480af:   8b 45 0c                mov    0xc(%ebp),%eax
 80480b2:   03 45 08                add    0x8(%ebp),%eax
 80480b5:   c9                      leave
 80480b6:   c3                      ret
```

Inline assembly (1/2)

- Basic inline assembly
  - asm(“statements”)
    - If a crash between string ‘asm’ and different name occurs, use __asm__ instead of asm. (same in case of ‘volatile’)
    - Insert “new_line” and “tab”(₩t) in each end of instruction selectively when use multiple instruction. The reason is to include assembly instructions to .s file generated by GCC according to format.
  - Example
    - asm(“nop”);, asm(“sti”);
    - asm(“movl $0x50, %eax\n\t”
      “addl %eax, %ebx”);
  - Problem
    - GCC can’t know change of resister value occurred by instruction in inline assembly.
    - In other word, problem of resister usage can be occurred. In the above example, resister eax and ebx can be spoiled.

Inline assembly (2/2)

- Extended inline assembly
  - asm (“statements”

```
    : output_list              /* optional */ → output C variables
    : input_list               /* optional */ → input C variables
    : overwrite_registers      /* optional */ → Modified registers
    )
```

  - Write C variables which will be output(write) in “output_list”. Compiler bind resister selectively.
  - Write C variables which will be input(read) in “input_list”. Compiler bind resister selectively.
  - “overwrite_registers” means writing resister declared that will be modified.
- Example – Use assembly code in C program

```
asm (“setae %%bl; movzbl %%bl, %0”
  : “=r” (result)  /* output */
  :                /* no inputs */
  : “%ebx”         /* overwrites */ )
```

  - Compiler(gcc) decide input/output resister. It is denoted by resister code used by gcc.
    - [resister] a: eax, b: ebx, c: ecx, d: edx, S: esi, D: edi, A: edx(MSB) ∪ eax(LSB)
    - [constant] I (capital i): 0 ~ 31 사이의 상수값 (for 32-bit shifts), K: 0xff, L: 0xffff
    - [dynamic allocation] q: one of {eax, ebx, ecx, edx}, r: one of {eax, ebx, ecx, edx esi, edi}, g: one of {eax, ebx, ecx, edx, memory/immediate operand}
    - [memory] m: memory operand

Inline assembly (1/2)

- Example 1

```
asm ("leal $3(%1,%1,2), %0"
        : "=r" (y)  /* '=' means "write-only" as a output. */
        : "r" (x)   /*  y = x*3 + 3 */
```

  - '%1' means '1'th(second) operand in whole operand list and indicate "r" (x).
  - Also, %0 means '0'th(first) and indicate "=r" (y).
  - In above example, changing with "x = x*3 + 3"

```
asm ("leal $3(%0,%0,2), %0"
        : "=r" (x)
        : "0" (x)   /* '0' means '0'th operand. */
```

- Example 2

```
__asm__ __volatile__ ("incl %0; sete %1"
                    : "=m" (xim), "=q" (cond)
                    : "m" (xim)
                    : "memory" )
```

  - 'volatile' means let it be in current position when compile this assembly code. (In case of volatile, unknown change can be occurred in compiler level (For example, state change of device register), so compiler don't optimize it but leave.)
  - 'memory' of overwrite list indicate change of memory value.

Inline assembly (2/2)

```
void main()
{
    int x=10, y=20, s;
    asm volatile ("movl  %1, %0\n\t"
                  "addl  %2, %0"
                  : "=r" (s)
                  : "r" (x), "r" (y)
                 );
}
```

gcc –O –S add_r.c

```
main:
            pushl       %ebp
            movl        %esp, %ebp
            subl        $8, %esp
            andl        $-16, %esp
            subl        $16, %esp
            movl        $10, %eax
            movl        $20, %edx
#APP
            movl  %eax, %eax
            addl  %edx, %eax
#NO_APP
            leave
            ret
```

```
void main()
{
    int x=10, y=20, s;
    asm volatile ("movl  %1, %0\n\t"
                  "addl  %2, %0"
                  : "=r" (s)
                  : “g" (x), “g" (y)
                 );
}
```

gcc –O –S add_g.c

```
main:
            pushl       %ebp
            movl        %esp, %ebp
            subl        $8, %esp
            andl        $-16, %esp
            subl        $16, %esp
#APP
            movl  $10, %eax
            addl  $20, %eax
#NO_APP
            leave
            ret
```

Code in #APP – #NO_APP : inline assembly code