GNU ASSEMBLER

Compilation

```
> gcc -S -m32 example.c

> gcc -m32 example.s

> ./a.out
```

```
#include <stdio.h>

main()
{
   int x ;
   int y ;

   x = 2*3 ;
   y = x + x ;
}
```

example.c

```
        .file   "example.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $16, %esp
        movl    $6, -4(%ebp)
        movl    -4(%ebp), %eax
        addl    %eax, %eax
        movl    %eax, -8(%ebp)
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
        .section        .note.GNU-stack,"",@progbits
```

example.s

AT&T Assembly Syntax(1)

- Format

```
operation source,destination
```

- Example: move the hexadecimal value 2 into the register al.

```
movb $0x02,%al
```

AT&T Assembly Syntax(2)

- AT&T immediate operands are preceded by $
- AT&T register operands are preceded by %
- AT&T absolute (as opposed to PC relative) jump/call operands are prefixed by *
- AT&T and Intel syntax use the opposite order for source and destination operands. Intel `add eax, 4' is `addl $4, %eax'. The `source, dest' convention is maintained for compatibility with previous Unix assemblers.
- In AT&T syntax the size of memory operands is determined from the last character of the opcode name. Opcode suffixes of `b', `w', and `l' specify byte (8-bit), word (16-bit), and long (32-bit) memory references.
- The AT&T assembler does not provide support for multiple segment programs. Unix style systems expect all programs to be single segment

Register Naming

- **32-bit registers:** `%eax' (the accumulator), `%ebx', `%ecx', `%edx', `%edi', `%esi', `%ebp' (the frame pointer), and `%esp' (the stack pointer).
- **16-bit low-ends of these:** `%ax', `%bx', `%cx', `%dx', `%di', `%si', `%bp', and `%sp'.
- **8-bit registers:** `%ah', `%al', `%bh', `%bl', `%ch', `%cl', `%dh', and `%dl' (These are the high-bytes and low-bytes of `%ax', `%bx', `%cx', and `%dx')

Memory References

An Intel syntax indirect memory reference of the form

[base + index*scale + disp]

is translated into the AT&T syntax

disp(base, index, scale)

Entering a Function

- When entering a function, the following operations are invoked

```
pushl  %ebp

movl   %esp, %ebp
```

Note that esp is pushed PC value (4 bytes)

Returning from a Function

- When returning from a function, leave operation invoked

```
leave
```

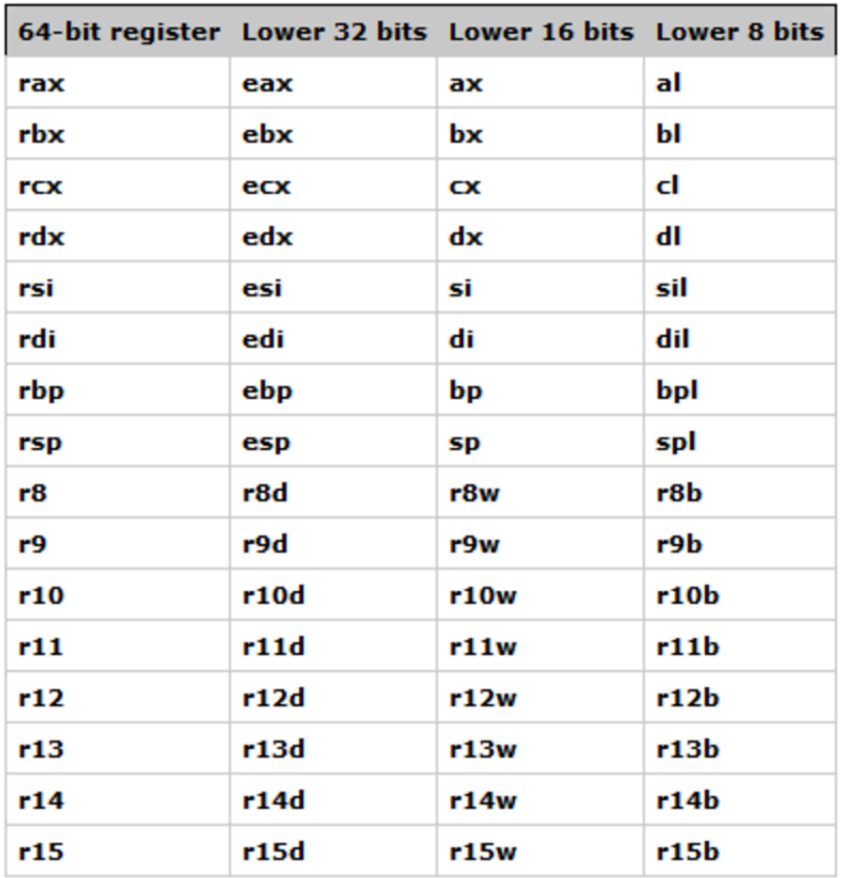
is equivalent to

```
movl  %ebp,$%esp
popl  %ebp
```

Showing Contents of Registers in the gdb debugger

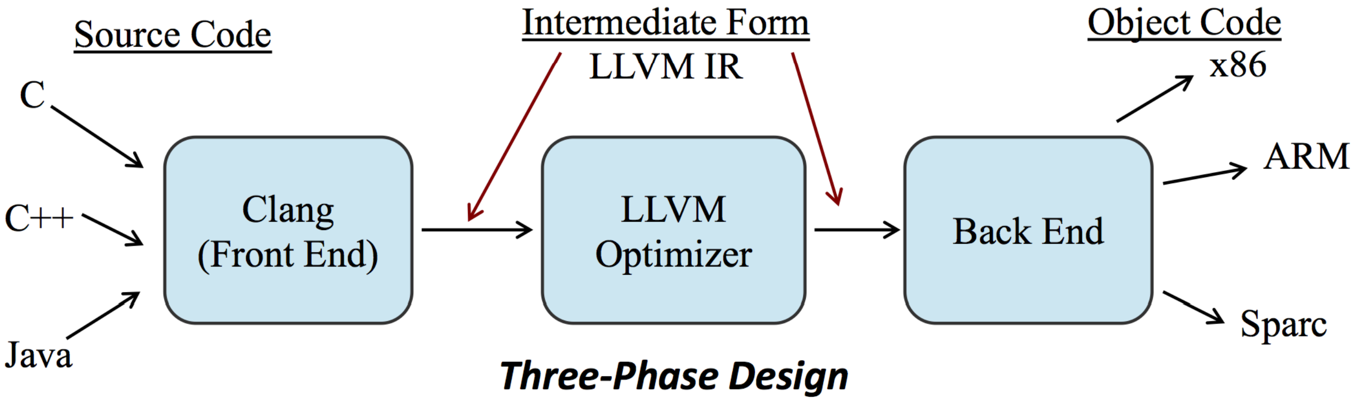
```
>info registers
```

```
>info registers eax
```



64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

LLVM Compiler Infrastructure



- To generate LLVM intermediate code (IR)

  >llvm-gcc example.c -S -emit-llvm

  or using clang

  >clang -emit-llvm –S example.c

- To run the program:

  >lli example.s

LLVM Example

- C Code

```
unsigned square_int(unsigned a) {
   return a*a;
}
```

- LLVM IR (intermediate representation)

```
define i32 @square_unsigned(i32 %a) {
   %1 = mul i32 %a, %a
    ret i32 %1
}
```