Machine-level Programming (1)

Why Machine-Level Programming?

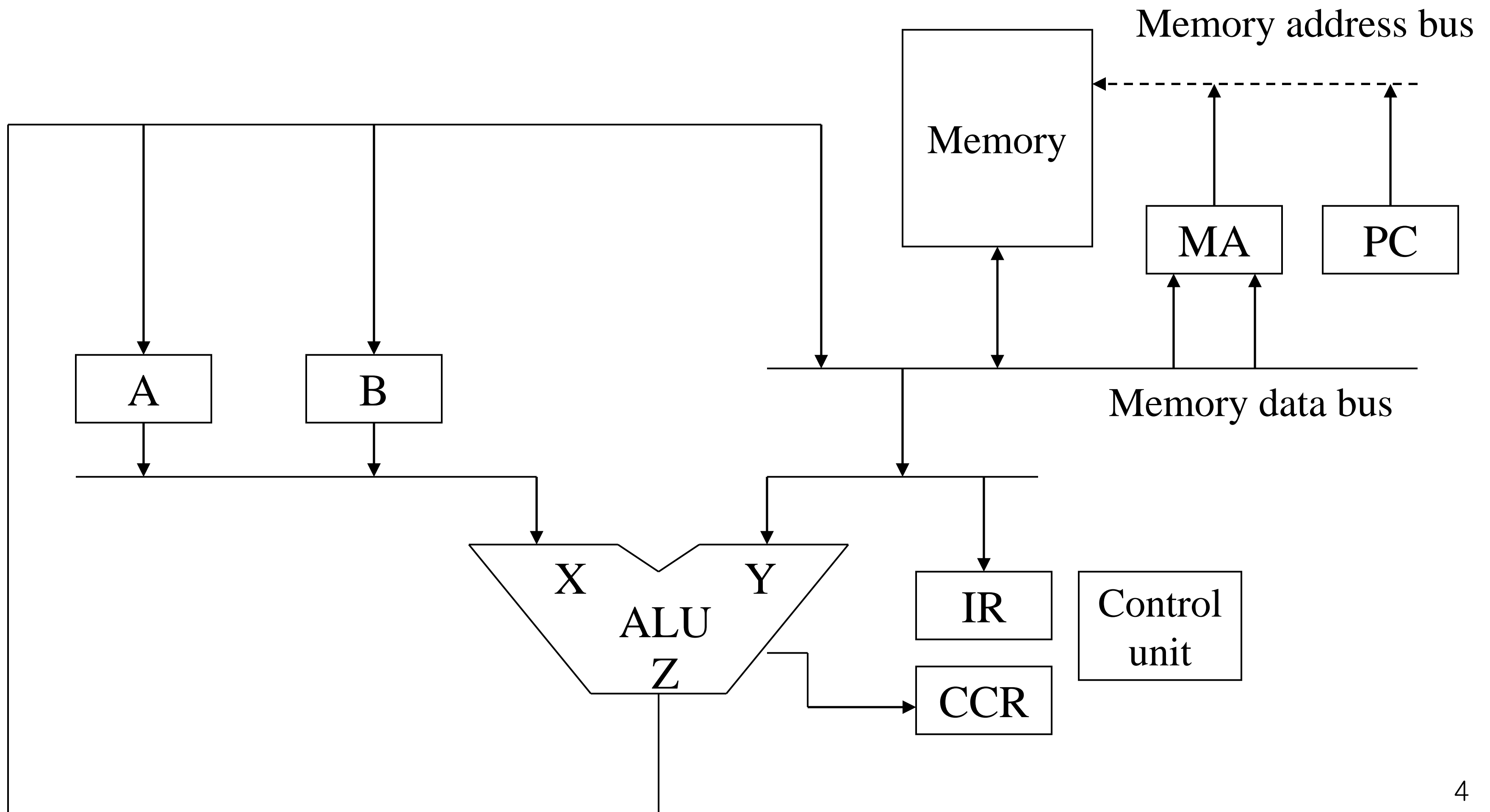
- Timing & Space critical programs (device drivers, OS)
  - Easy prediction of timing and space
- Easy access of HW (Difficult in C-level)
- Use extensive instructions (while GCC does not)
- Gives comprehensive understanding of interactions between HW and SW

어셈블리어에의 접근방법

- 어셈블리 프로그램의 활용
  - 빠른 실행이 요구되는 코드 부분
  - 고급언어로는 작성하기 어려운 코드 부분
  - 실행 코드의 성능 및 동작 분석
- 문제점
  - 어셈블리 프로그램의 작성능력을 배양하는데는 많은 시간과 노력이 요구됨
  - 해결방안(예)
    - C 언어로 작성하여 컴파일한 후, 디스어셈블리 과정을 통하여 어셈블리 (혹은 기계어) 코드를 출력으로 생성
    - 혹은, 기존의 코드를 디스어셈블리를 통하여 어셈블리 코드 생성
  - 이러한 코드를 분석하여 최적화하거나 수정함.
- 디스어셈블리 (disassembly)
  - 실행가능한 코드(이진 코드)를 검사분석하여 어셈블리어로 재구성하는 것. 즉, 어셈블리 과정의 정반대 과정(reverse process).
  - 디스어셈블러(disassembler)에 의해 구현됨.
- Linux에서의 어셈블리어 분석방법
  - C 프로그램을 어셈블리어 코드로 변환: .s 파일의 생성
    - Linux command: gcc –O –S code_c2_addressing.c
  - .s 파일을 어셈블하여 실행코드(executable code)로 변환: a.out 파일의 생성
    - Linux command: as code_c2_addressing.s
  - 목적코드를 디스어셈블함: 어셈블리 코드의 생성
    - Linux command: objdump –d a.out”

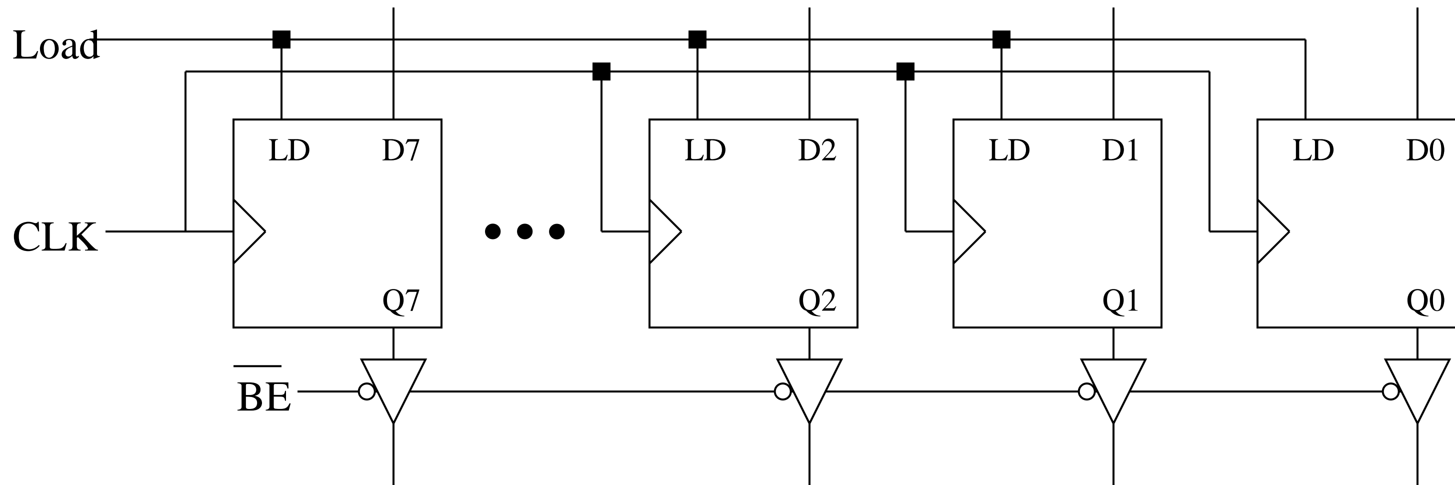
Simplified Processor Structure

- Executing an machine instruction is moving around data among registers and memory through ALU
- Example
  - 0x08101000: add $10 %B



Structure of a register

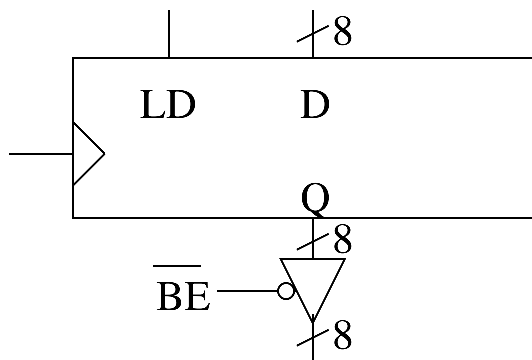
- Load data in off a bus
- Connect register output to a bus
- Consider a 8-bit register

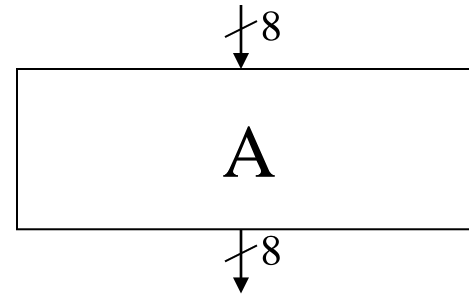


- How to load or hold?
- How to connect to bus (need a way to connect/disconnect)
  → Tri-state buffer

Simplified Symbols

Simplified symbols





Instruction Set Architecture (ISA)

- What a machine-level (assembly-level) programmer needs to know?
  - Architecture = registers + instruction set

1. set of instructions (opcodes)
2. Include addressing mode
   - Ways instructions get data from memory
   - Ways to address memory

- Programmer's Model
  - What registers do you work with?
  - What operations are available?
- Thousands of ISA: but the same principle
- We will work with IA32 (Intel Architecture 32)

Intel Architecture 32-bit

- IA-32 Microprocessors
  - IA-32: Intel [instruction set] Architecture, 32-bit
  - Architecture for Intel 32-bit 80x86 microprocessor
  - backward-compatible to 16-bit 80x86
  - Dominant in PC Computer Market
- IA-32 Quick View
  - 32bit Registers
    - data registers: eax, ebx, ecx, edx
    - address registers: esp, ebp, esi, edi
    - condition code registers
    - instruction pointer register (PC)
  - Instructions
    - load/store
    - arithmetic operations
    - logical operations
    - branch/jump
    - procedure call and return

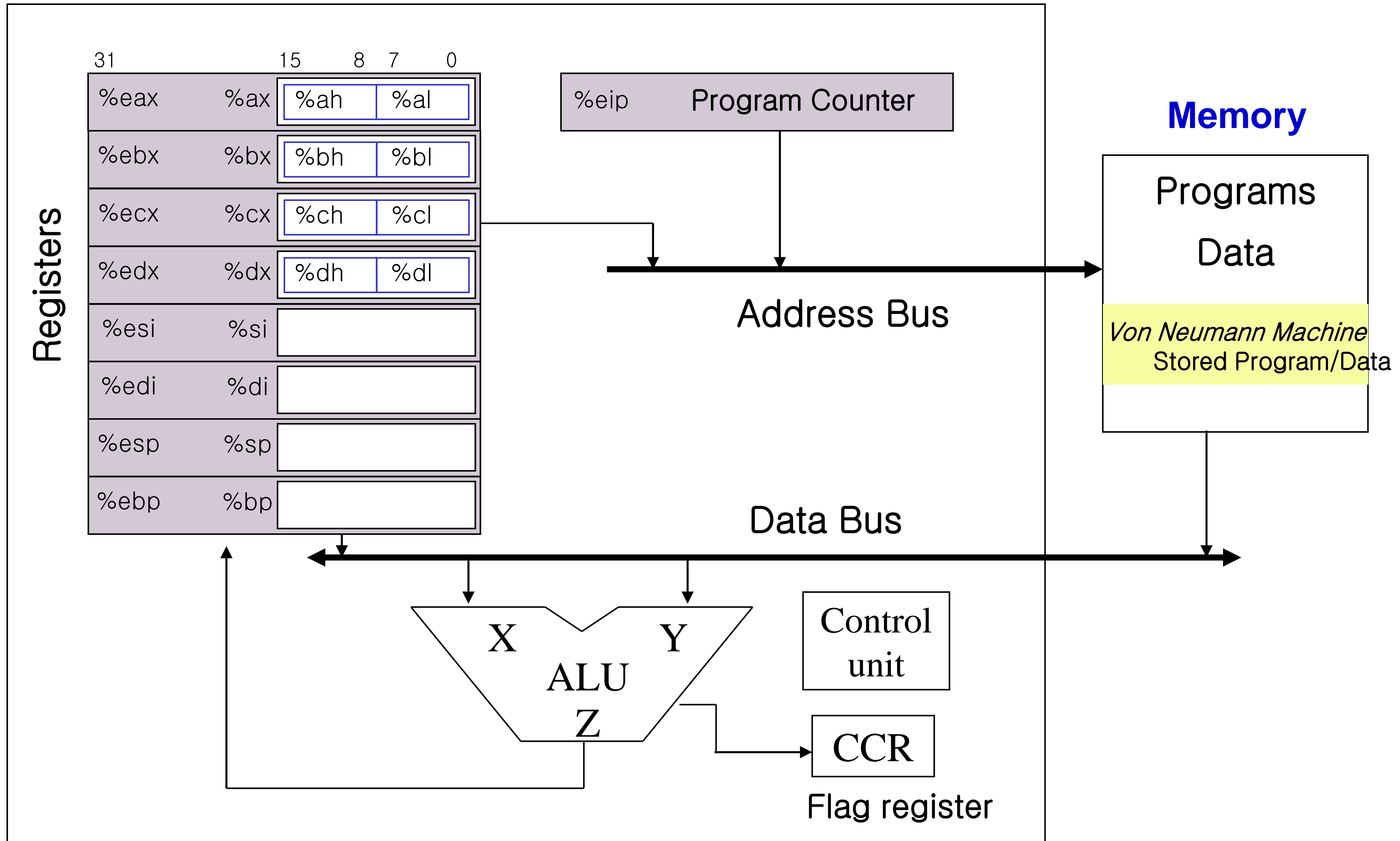
Intel 마이크로프로세서의 진화

- 4-bit
  - 4004 (1971년 11월): 세계 최초의 상용(商用) 단일 칩 마이크로프로세서
    - clock speed 740 kHz, 트랜지스터 2300 개, 프로그램 메모리 4 KiB, 0.06 MIPS
  - 4040 (1974)
- 8-bit
  - 8008 (1972), 8080 (1974), 8085(1976년)
- 16-bit
  - 8086 (1978): 80x86 마이크로프로세서의 원조
    - 최대 clock speed 10 MHz (0.75 MIPS), 트랜지스터 29,000 개
    - 데이터 버스 16 비트, 주소 20 비트
    - 8088 (1979): 외부로의 데이터 버스 8 비트. IBM PC에 사용됨.
  - 80186 (1982, 내장형 응용), 80286 (1982)
- 32-bit
  - 80386 (1985): 최초의 32 비트 마이크로프로세서
    - 최대 clock speed 33 MHz (11.4 MIPS), 트랜지스터 275,000 개
    - 데이터 버스, 주소 버스 각 32 비트. 주소공간 4 GiB
  - 80486 (1989), Pentium (1993), Pentium Pro (1995), Pentium II (1997), Pentium III (1999), Pentium 4 (2000)

[註|例] Intel 마이크로프로세서의 진화

μP	레지스터 크기 (bits)	데이터 버스 (bits)	주소공간(bytes)	트랜지스터 수	비고
8086	16	16	1M	29K	real mode, 1978년
8088	16	8	1M	29K	real mode, 1979
80286	16	16	16M	134K	real/protected mode, 1982
80386	32	32	4G	275K	flat addressing, 1985
80486	32	32	4G	1.9M	1989
Pentium	32	64	4G	3.1M	superscalar, 1993
Pentium II	32	64	4G	7.5M	1997
Pentium III	32	64	4G	8.2M	1999
Pentium 4	32	64	4G	42M	2000
Itanium 2	64	128	(64-bit)	221M	2002, for 64-bit computing, on-board “x86 engine”
Intel Core 2	64	64	(64-bit)	291M (4MiB L2)	2006. 80x86-binary-compatible 2 CPUs in one package

System Programmer's view of 80x86

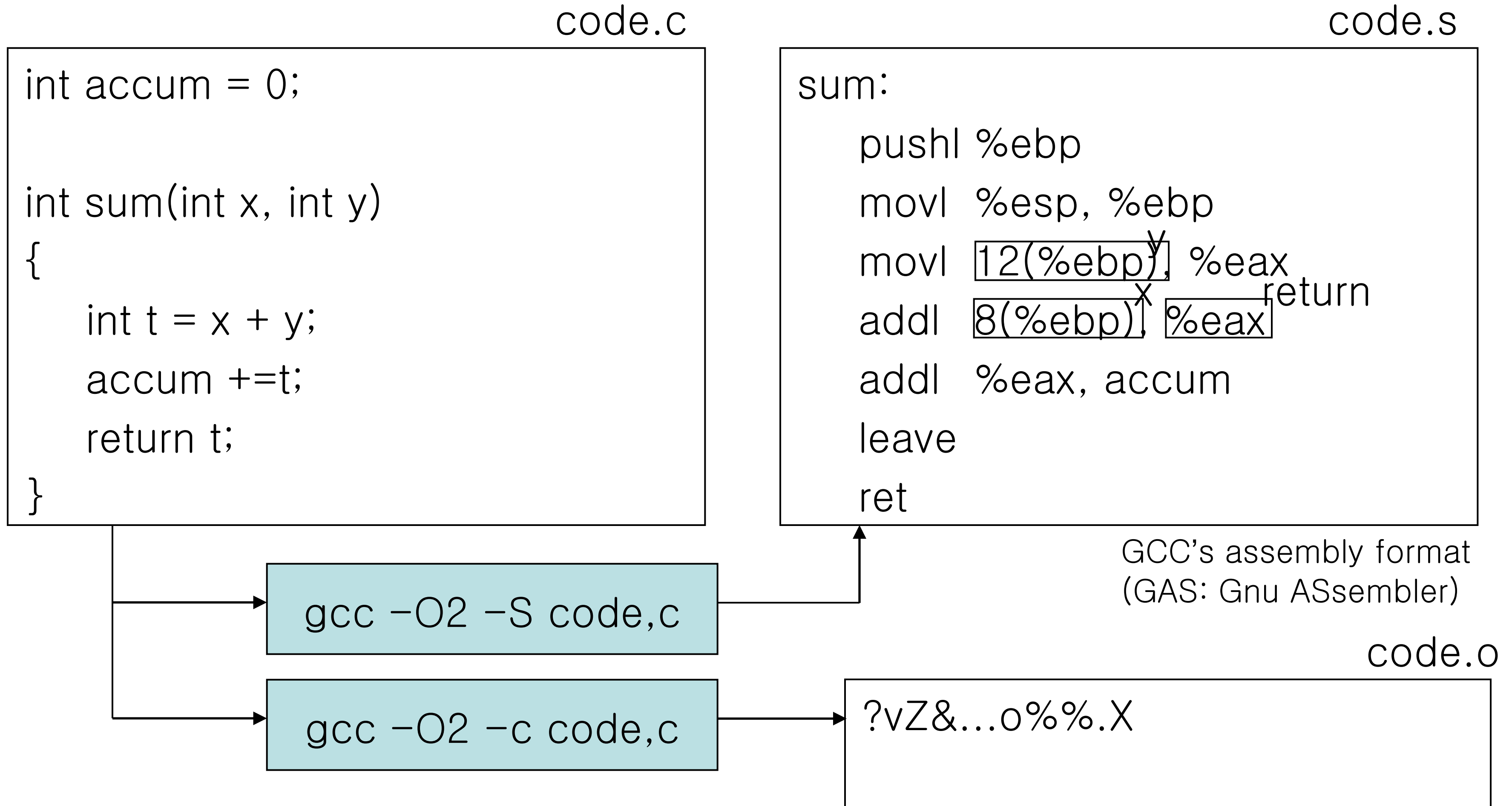


C program vs. Assembly program

- In C
  - Different data types (char, signed, unsigned)
  - Various constructs (if-else, for, while, switch, procedure call)
- In Assembly
  - No distinction (byte or byte array)
  - Only elementary operations (add, move, conditional jump)

Understanding assembly code and how it relates to the original C code is a key step in understanding how computers execute programs

Code Example



Dissassembler

code.o

?vZ&...o%%.X........

objdump –d code.o

```
00000000 <sum>:
   0: 55                  push   %ebp
   1: 89 e5               mov    %esp,%ebp
   3: 8b 45 0c            mov    0xc(%ebp),%eax
   6: 03 45 08            add    0x8(%ebp),%eax
   9: 01 05 00 00 00 00   add    %eax,0x0
   f: c9                  leave
  10: c3                  ret
```

gcc –O2 –o prog code.o main.c

prog

xwO*@R..%$....

objdump –d prog

```
080482f4 <sum>:
 80482f4:  55                   push   %ebp
 80482f5:  89 e5                mov    %esp,%ebp
 80482f7:  8b 45 0c             mov    0xc(%ebp),%eax
 80482fa:  03 45 08             add    0x8(%ebp),%eax
 80482fd:  01 05 74 93 04 08    add    %eax,0x8049374
 8048303:  c9                   leave
 8048304:  c3                   ret
 8048305:  90                   nop
 8048306:  90                   nop
 8048307:  90                   nop
```

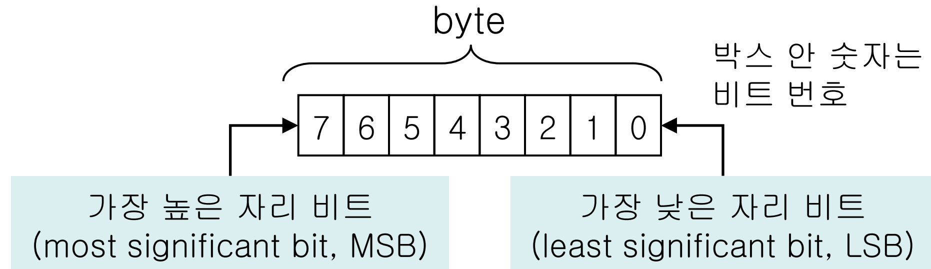
main.c

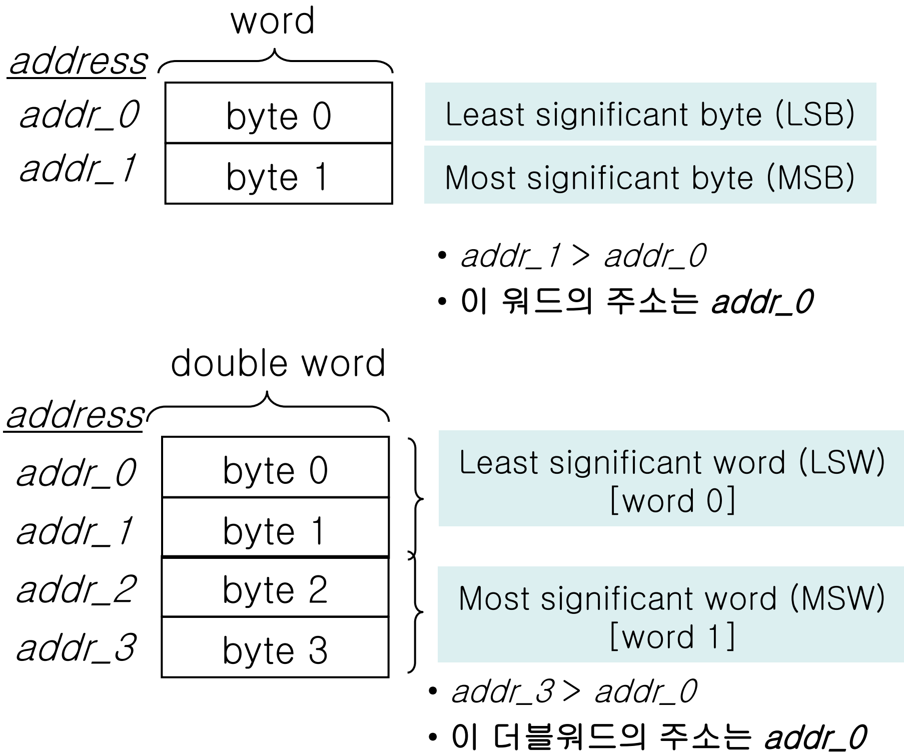
```
int main()
{
    return sum(1,3);
}
```

[註|例] 데이터 유형

- Little endian vs. big endian
  - 문제점: 두개의 바이트 이상을 차지하는 정수를 메모리에 저장할때, 어느 바이트에 윗자리 수를 저장할까.
  - 예를 들면 0x8452를 저장할때 두 바이트가 필요함. 0x84 한 바이트, 0x52 한 바이트.
    - 이것을 100번지, 101번지에 저장한다고 하면, 일단 이 정수의 위치(즉, 주소)는 100번지가 됨. 101번지는 될 수 없음.
    - 그러면, 윗자리 수인 0x84를 100번지에 저장할까, 아니면 101번지에 저장할까? 이것이 문제로다.
    - 100 번지에 0x84를 저장하면 윗자리수가 먼저 오니까 big endian (물론 101번지에 0x52가 저장)
    - 100 번지에 아랫자리수인 0x52가 오면 little endian (물론 101번지에 0x84가 저장됨)
  - 암기 요령: 윗자리 → big, 아랫자리 → little

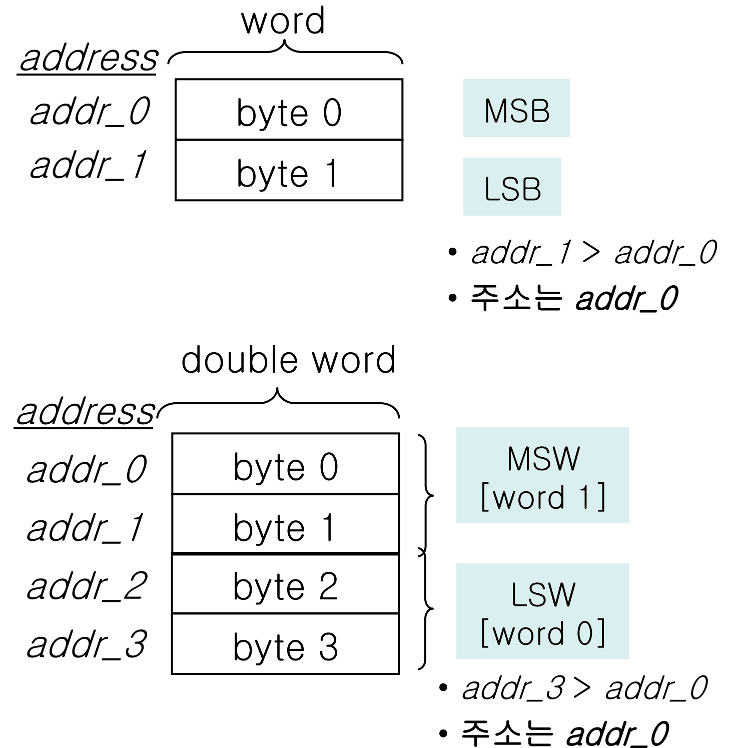
[註|例] 데이터 유형





- *addr_1* > *addr_0*
- 이 워드의 주소는 ***addr_0***

- *addr_3* > *addr_0*
- 이 더블워드의 주소는 ***addr_0***



- *addr_1* > *addr_0*
- 주소는 ***addr_0***

- *addr_3* > *addr_0*
- 주소는 ***addr_0***

Data Formats

Machine representation for C data types: no distinction except no. of bytes

C declaration	Intel data type	GAS suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
unsigned	Double word	l	4
long int	Double word	l	4
unsigned long	Double word	l	4
char *	Double word	l	4
float	Single precision	s	4
double	Double precision	l	8
long double	Extended precision	t	10/12

Even for 32-bit machine, due to its origins as a 16-bit architecture that expanded into a 32-bit one, Intel uses the term “word” to refer to a 16-bit data type. “double word”=32-bits, “quad word” = 64-bits.

Note: GAS suffix

mov: movb (move byte), movw (move word), movl (move double word)

Instruction Formats

- An instruction consists of
  - opcode + operands
  - *operation code, opcode*
    - Indicates operation type, ex) add, subtract, move, halt
  - *operands*
    - data with which the operation is performed
    - usually one or more operands: source, destination
    - several ways to indicate the data (operand forms)
      - Immediate
      - Register
      - Memory: several ways to point the memory location (addressing mode)
        - » Absolute
        - » Indirect
        - » Base + Displacement
        - » Indexed
        - » Scaled Indexed

Operand Forms and Addressing Modes

Type	Example	Form	Operand value	Name
Immediate	$0x10	$Imm	Imm	Immediate
Register	%eax	Ea	R[Ea]	Register
Memory	0x080934	Imm	M[Imm]	Absolute
Memory	(%eax)	(Ea)	M[R[Ea]]	Indirect
Memory	8(%eax)	Imm(Eb)	M[Imm+R[Eb]]	Base+displacement
Memory	(%eax,%edx)	(Eb, Ei)	M[R[Eb]+R[Ei]]	Indexed
Memory	8(%eax,%edx)	Imm(Eb, Ei)	M[Imm+R[Eb]+R[Ei]]	Indexed
Memory	(, %ecx, 4)	(, Ei, s)	M[R[Ei]*s]	Scaled indexed
Memory	8(, %ecx, 4)	Imm (, Ei, s)	M[Imm+R[Ei]*s]	Scaled indexed
Memory	(%eax,%edx,4)	(Eb,Ei,s)	M[R[Eb]+R[Ei]*s]	Scaled indexed
Memory	8 (%eax,%edx,4)	Imm (Eb,Ei,s)	M[Imm+R[Eb]+R[Ei]*s]	Scaled indexed

→ Addressing Modes: Generic Form

Imm(Eb,Ei,s): effective address=Imm+R[Eb]+R[Ei]*s

Operand Forms and Addressing Modes (Examples)

Assume

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in

Operand	Effective Address	Value
%ecx		
0x104		
$0x108		
(%eax)		
4(%eax)		
9(%eax,%edx)		
260(%ecx,%edx)		
0xFC(, %ecx,4)		
(%eax,%edx,4)		

Data Movement Instructions

Instruction	Effect	Description

movl S, D
movw S, D
movb S,D
	D ← S
	D ← S
	D ← S
		Move double word
		Move word
		Move byte

movsbl S, D movzbl S, D	D ← SignExtend(S) D ← ZeroExtend(S)	Move sign-extended byte Move zero-extended byte
pushl S popl D	R[%esp] ← R[%esp]-4; M[R[%esp]] ← S D ← M[R[%esp]]; R[%esp] ← R[%esp]+4	Push Pop

Example of Move Instructions

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

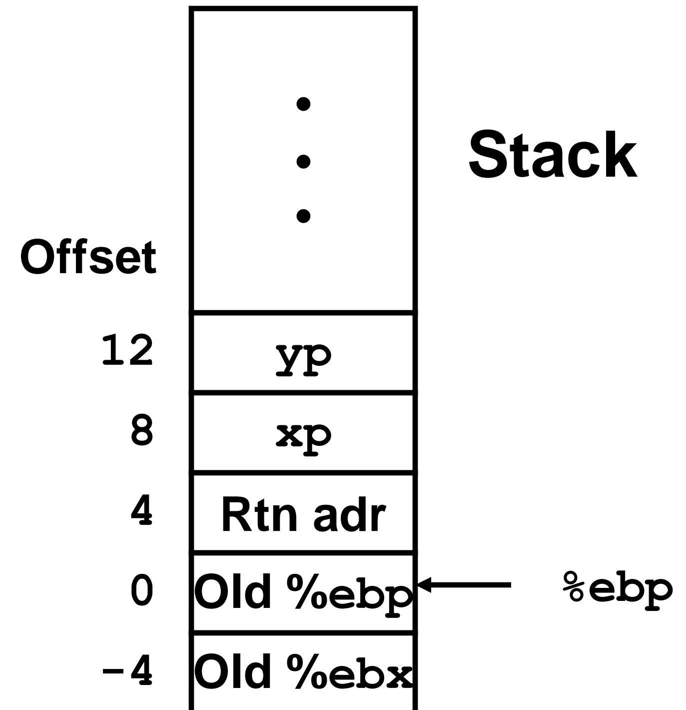
Set Up

Body

Finish

Understanding Swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

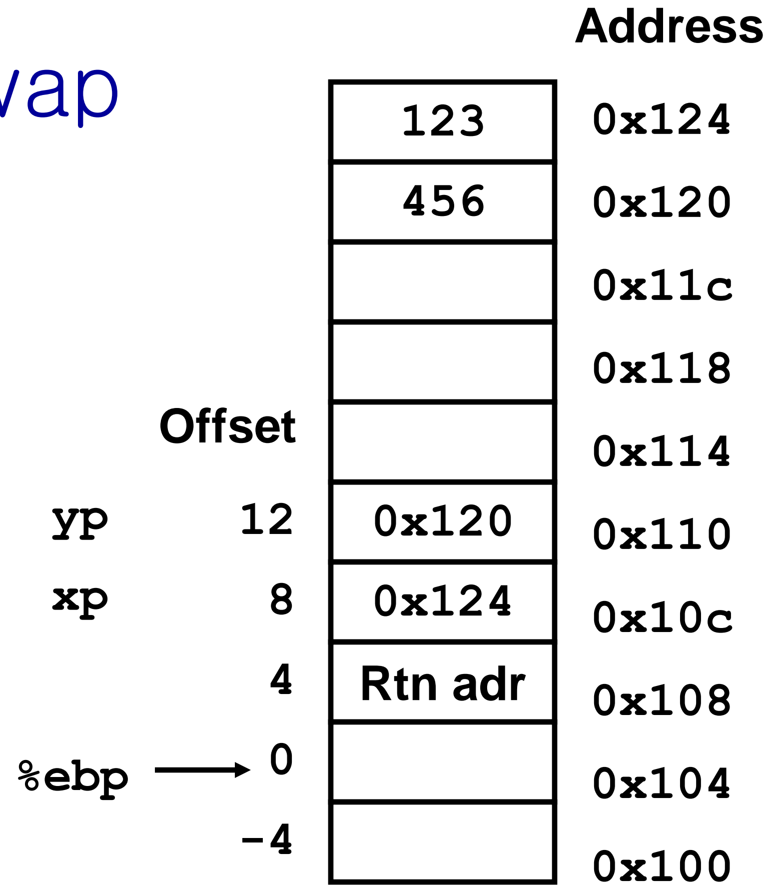


Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap

%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap

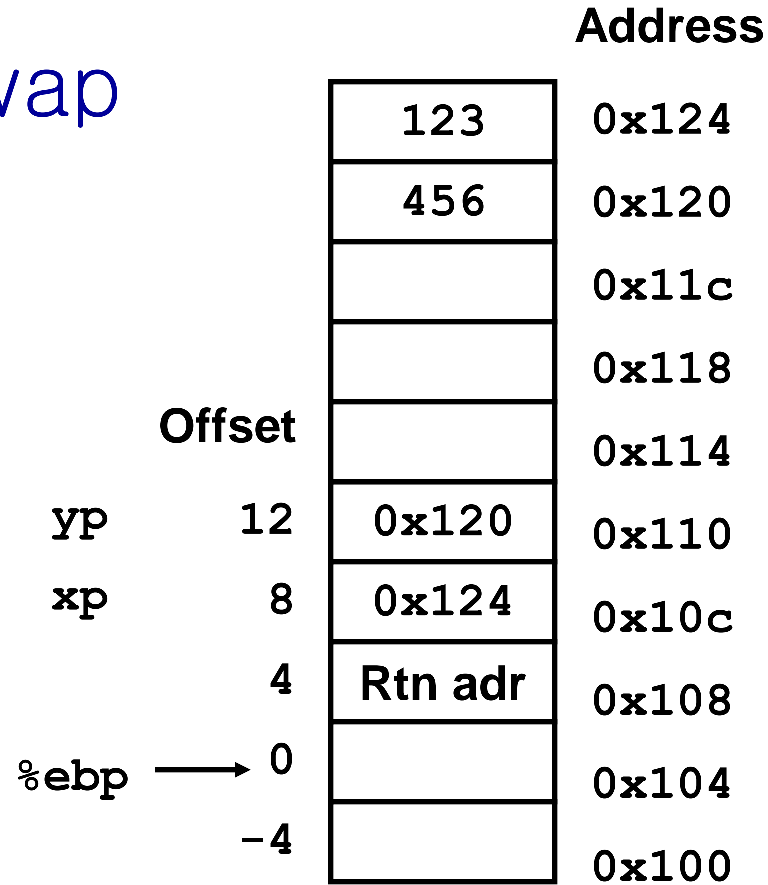
Register	Value
%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

	Offset	Memory	Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap

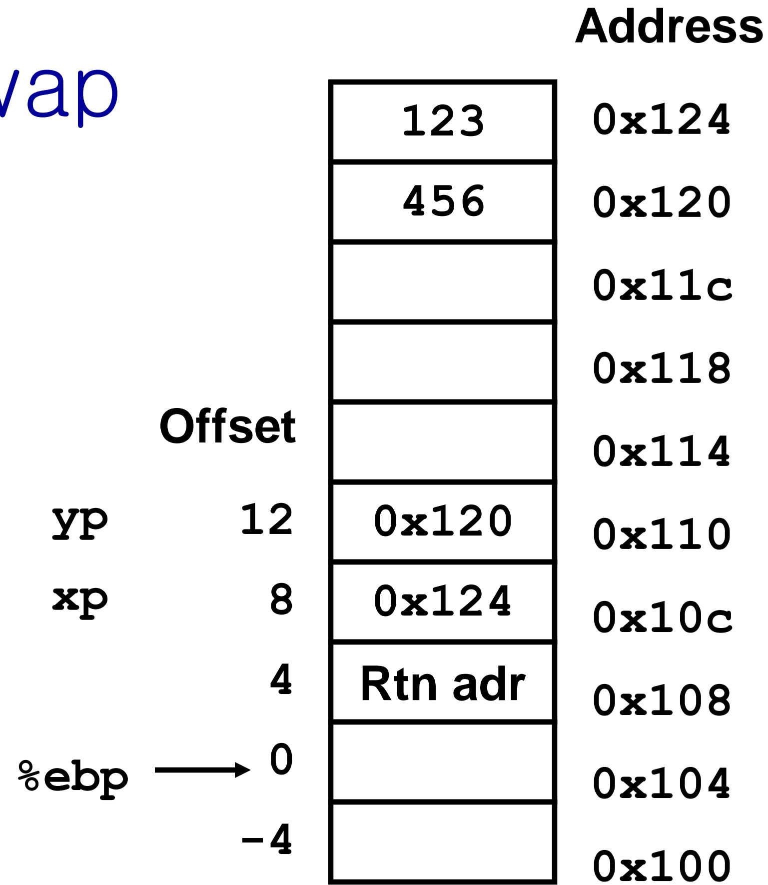
%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

Understanding Swap

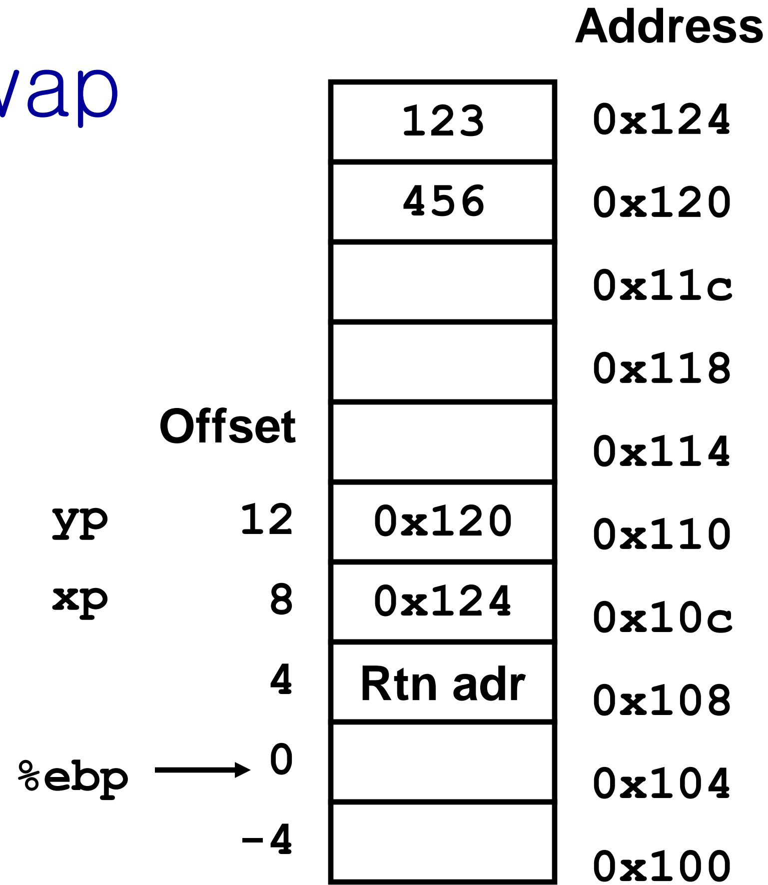
%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx    # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

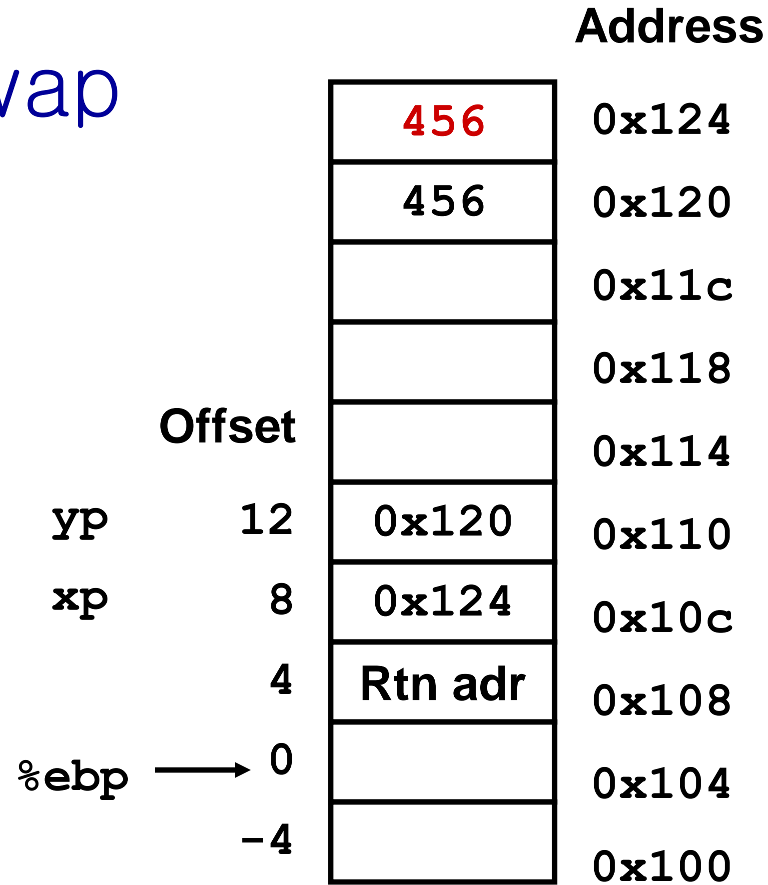
Understanding Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap



Register	Value
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Understanding Swap

Register	Value
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104

	Offset	Memory	Address
		456	0x124
		123	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Move (Sign/Zero) Extended Byte

k-bit number: $b_{k-1}\ b_{k-2}\ \ldots.\ b_1\ b_0$
Unsigned Interpretation:
$2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \ldots + 2^1b_1 + 2^0b_0$

Signed 2's Complement Interpretation
$-2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \ldots + 2^1b_1 + 2^0b_0$

```
movsbl $0x81 %eax
movzbl $0x81 %eax
```

Think about (k+1)-bit number (what should be b_k?)

Value	Interpretation	
	Unsigned	Signed 2's Comp.
0x81 (10000001)	129	-127

⇩ Extension to 16 bits?

0x0081 (00000000 10000001)	129	
0xFF81 (11111111 10000001)		-127

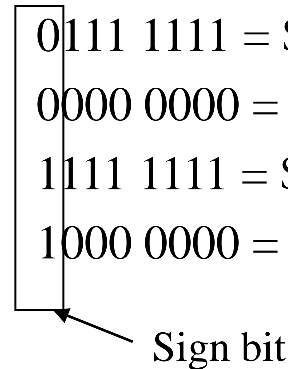
⇩ Extension to 32 bits?

Unsigned or Signed Numbers

- No distinction in machine-level
  - Unsigned or Signed (2's complement form) are programmer's interpretation

- Unsigned 8-bit number
  - 0000 0000 = $00 = 0
  - 1111 1111 = $FF = 255

- Signed 8-bit number
  - 0111 1111 = $7F = 127
  - 0000 0000 = $00 = 0
  - 1111 1111 = $FF = -1
  - 1000 0000 = $80 = -128



Signed 2's complement

- Signed 2's complement
  - MSB – sign bit
  - Negative numbers – 2's complement of absolute value
- How to make 2's complement of an absolute value
  - method A) 1's complement +1
    - −1= −(0000 0001) = 1111 1110 + 1 = 1111 1111
  - method B) scan from right to left
    - Write down bits until first 1
    - Complement after
    - −(0000 0001) = 1111 1111
- Determine absolute value of negative numbers (ex. $8A)
  - 1000 1010 = −128+(8+2) = −118
  - 1000 1010 = −(01110110) = −118
  - What if $8A considered as an unsigned value: $8A = 128+10 = 138

Push and Pop

Initially

%eax	0x123
%edx	0
%esp	0x108

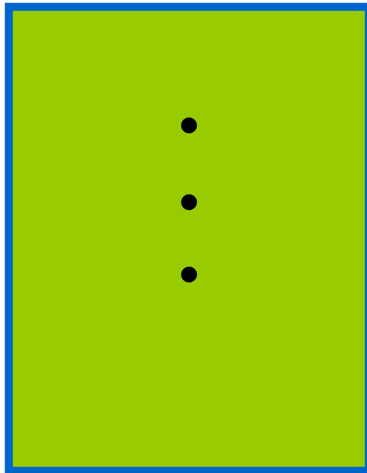
`pushl %eax`

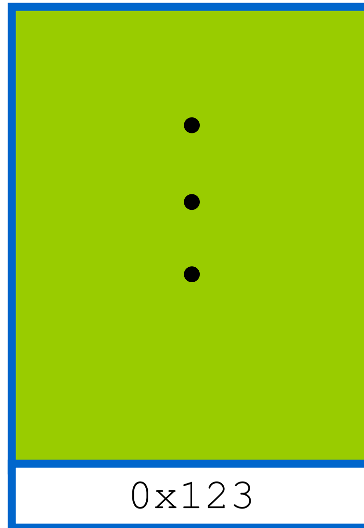
%eax	0x123
%edx	0
%esp	0x104

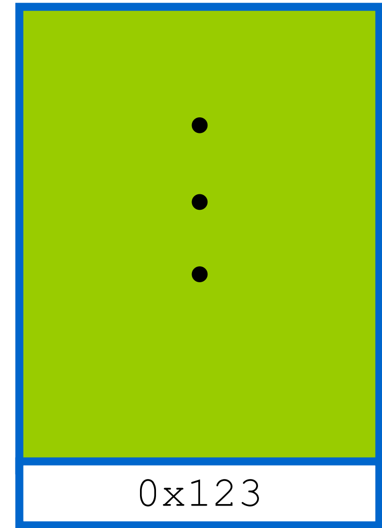
`popl %edx`

%eax	0x123
%edx	0x123
%esp	0x108

Increasing address







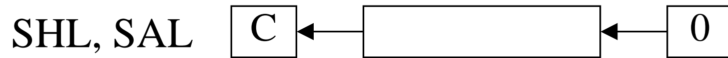
Arithmetic and Logical Operations

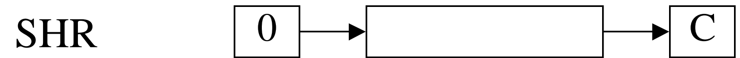
Instruction	Effect	Description
leal S,D	D ← &S	Load effective address
incl D	D ← D+1	Increment
decl D	D ← D−1	Decrement
negl D	D ← −D	Negate
notl D	D ← ~D	Complement
addl S,D	D ← D + S	Add
subl S,D	D ← D − S	Subtract
imull S,D	D ← D * S	Multiply
xorl S,D	D ← D ^ S	Exclusive-or
orl S,D	D ← D | S	Or
andl S,D	D ← D & S	And
sall k, D	D ← D << k	Left shift
shll k, D	D ← D << k	Left shift (same as sall)
sarl k, D	D ← D >> k	Arithmetic right shift
shrl k, D	D ← D >> k	Logical right shift

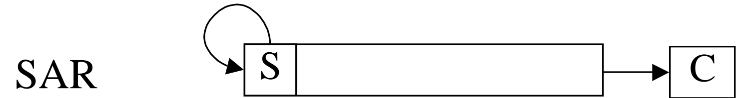
Address Computation Instruction

- `leal` *Src*,*Dest*
  - *Src* is address mode expression
  - Set *Dest* to “address” denoted by expression (NO actual read from that memory location)
- Two Uses
  - Computing address without doing memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8.

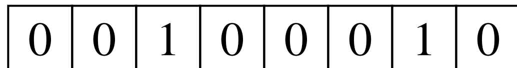
Arithmetic and Logical Shift





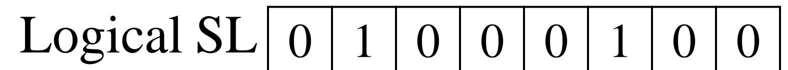


A= $22 = 34 unsigned



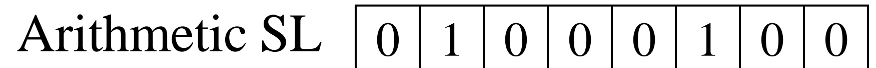
A= $22 = 34 signed

A= $44 = 68

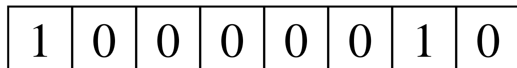


||

A= $44 = 68

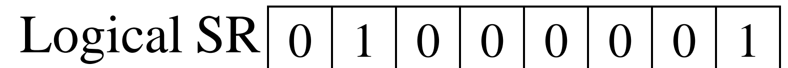


A= $82 = 130 unsigned

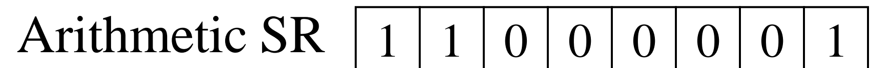


A= $82 = -126 signed

A= $41 = 65



A= $C1 = -63



Examples of Arithmetic Instructions

```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

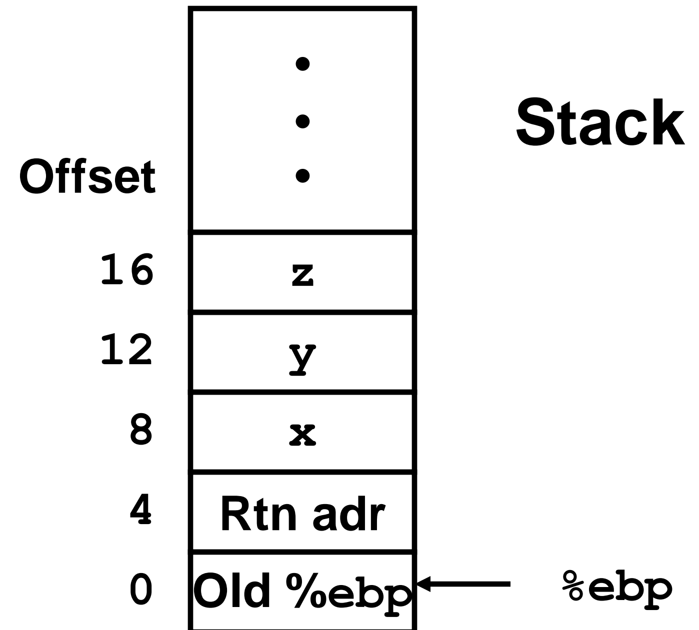
```
arith:
    pushl %ebp                    } Set
    movl %esp,%ebp                } Up

    movl 8(%ebp),%eax             }
    movl 12(%ebp),%edx            }
    leal (%edx,%eax),%ecx         }
    leal (%edx,%edx,2),%edx       }
    sall $4,%edx                  } Body
    addl 16(%ebp),%ecx            }
    leal 4(%edx,%eax),%eax        }
    imull %ecx,%eax               }

    movl %ebp,%esp                }
    popl %ebp                     } Finish
    ret                           }
```

Understanding `arith`

```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y  (t4)
addl 16(%ebp),%ecx         # ecx = z+t1  (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2  (rval)
```

Understanding `arith`

```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
# eax = x
  movl 8(%ebp),%eax
# edx = y
  movl 12(%ebp),%edx
# ecx = x+y  (t1)
  leal (%edx,%eax),%ecx
# edx = 3*y
  leal (%edx,%edx,2),%edx
# edx = 48*y (t4)
  sall $4,%edx
# ecx = z+t1 (t2)
  addl 16(%ebp),%ecx
# eax = 4+t4+x (t5)
  leal 4(%edx,%eax),%eax
# eax = t5*t2 (rval)
  imull %ecx,%eax
```

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp              } Set Up
    movl %esp,%ebp

    movl 8(%ebp),%eax       }
    xorl 12(%ebp),%eax      } Body
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp          }
    popl %ebp               } Finish
    ret
```

2^{13} = 8192, 2^{13} – 7 = 8185

```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y      (t1)
sarl $17,%eax          eax = t1>>17   (t2)
andl $8185,%eax        eax = t2 & 8185
```